# EXHIBIT C

U 7172608

# THE UNITED STATES OF AMERICA

## TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office

February 27, 2009

THIS IS TO CERTIFY THAT ANNEXED HERETO IS A TRUE COPY FROM THE RECORDS OF THIS OFFICE OF:

U.S. PATENT: *5,481,721*
ISSUE DATE: *January 02, 1996*

By Authority of the
Under Secretary of Commerce for Intellectual Property
and Director of the United States Patent and Trademark Office

P. SWAIN
Certifying Officer

US005481721A

# United States Patent [19]

Serlet et al.

[11] Patent Number: 5,481,721

[45] Date of Patent: Jan. 2, 1996

[54] **METHOD FOR PROVIDING AUTOMATIC AND DYNAMIC TRANSLATION OF OBJECT ORIENTED PROGRAMMING LANGUAGE-BASED MESSAGE PASSING INTO OPERATION SYSTEM MESSAGE PASSING USING PROXY OBJECTS**

[75] Inventors: **Bertrand Serlet**; **Lee Boynton**, both of Palo Alto; **Avadis Tevanian**, Mountain View, all of Calif.

[73] Assignee: **NeXT Computer, Inc.**, Redwood City, Calif.

[21] Appl. No.: **332,486**

[22] Filed: **Oct. 31, 1994**

**Related U.S. Application Data**

[63] Continuation of Ser. No. 731,636, Jul. 17, 1991, abandoned.

[51] **Int. Cl.**[6] ..... **G06F 9/44**

[52] **U.S. Cl.** ..... **395/700**; 364/DIG. 1; 364/280; 364/284.3; 364/284

[58] **Field of Search** ..... 395/700, 650; 364/DIG. 1, DIG. 2

[56] **References Cited**

U.S. PATENT DOCUMENTS

| | | | |
|---|---|---|---|
| 5,060,150 | 10/1991 | Simor | 364/200 |
| 5,230,051 | 7/1993 | Quan | 395/700 |
| 5,305,461 | 4/1994 | Feigenbaum et al. | 395/775 |

OTHER PUBLICATIONS

Bennet, J. K., "The design and implementation of Distributed Smalltalk", SIGPLAN Notices, vol. 22, No. 12, pp. 318–320, OOPSLA '87 Proceedings, Dec. 1987.

McCullough, P. L., "Transparent Forwarding: First Steps", SIGPLAN Notices, vol. 22, No. 12, pp. 331–341, OOPSLA '87 Proceedings, Dec. 1987.

Shapiro, M., "The Design of a Distributed Object–Oriented Operating System For Office Applications", ESPRIT '88. Putting the Technology to Use Proceedings of the 5th Annual ESPRIT Conference, pp. 1020–1027, vol. 2, Nov. 1988.

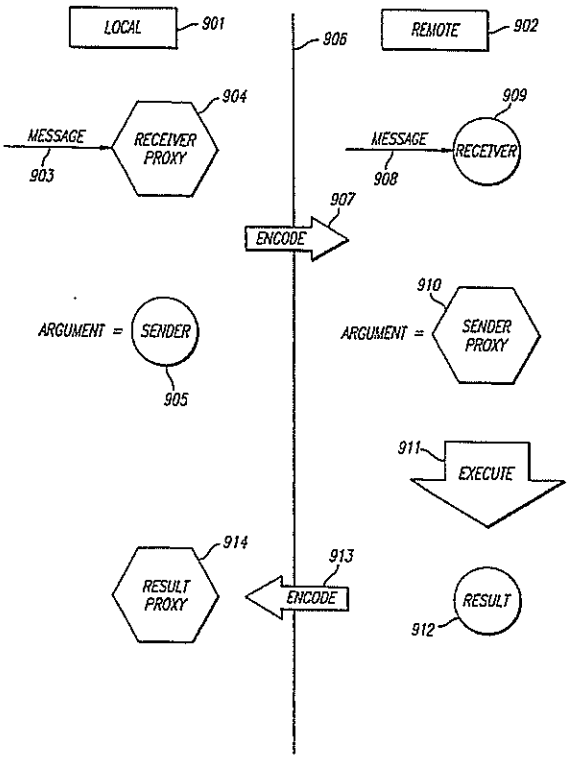
*Primary Examiner*—Kevin A. Kriess

*Attorney, Agent, or Firm*—Hecker & Harriman

[57] **ABSTRACT**

The present invention provides a method and apparatus for the distribution of objects and the sending of messages between objects that are located in different processes. Initially, a "proxy" object is created in the same process as a sender object. This proxy acts as a local receiver for all objects in the local program. When the proxy receives a message, the message is encoded and transmitted between programs as a stream of bytes. In the remote process, the message is decoded and executed as if the sender was remote. The result follows the same path, encoded, transmitted, and then decoded back in the local process. The result is then provided to the sending object.

**24 Claims, 6 Drawing Sheets**



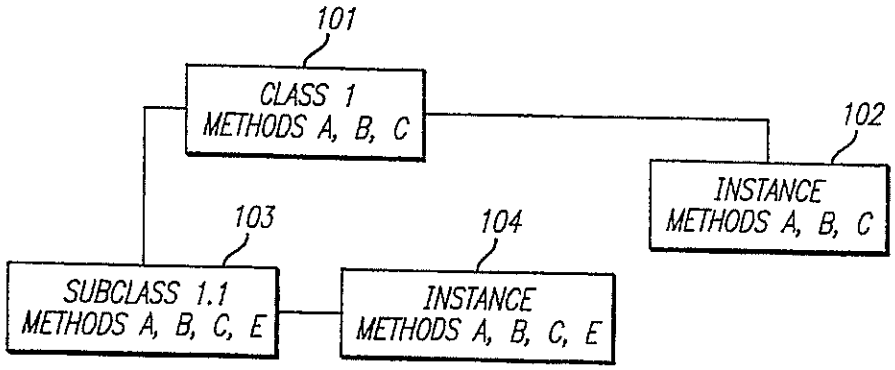



FIG. 1 PRIOR ART

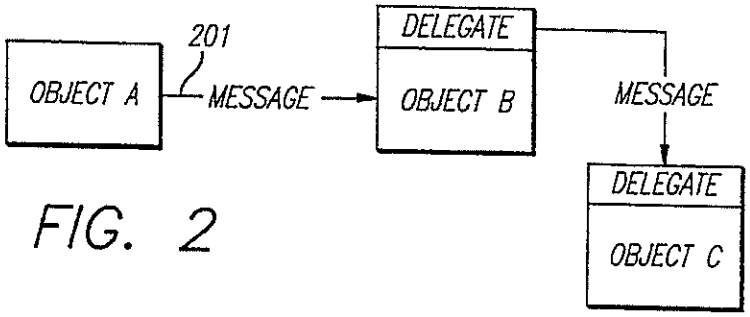



FIG. 2

FIG. 4

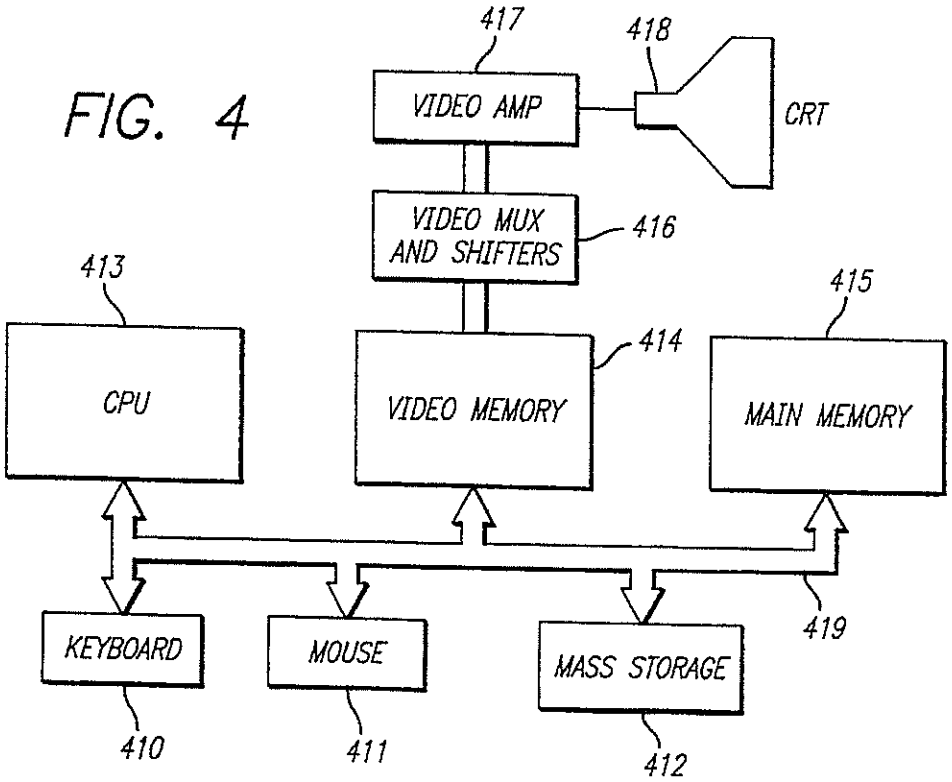


FIG. 3A

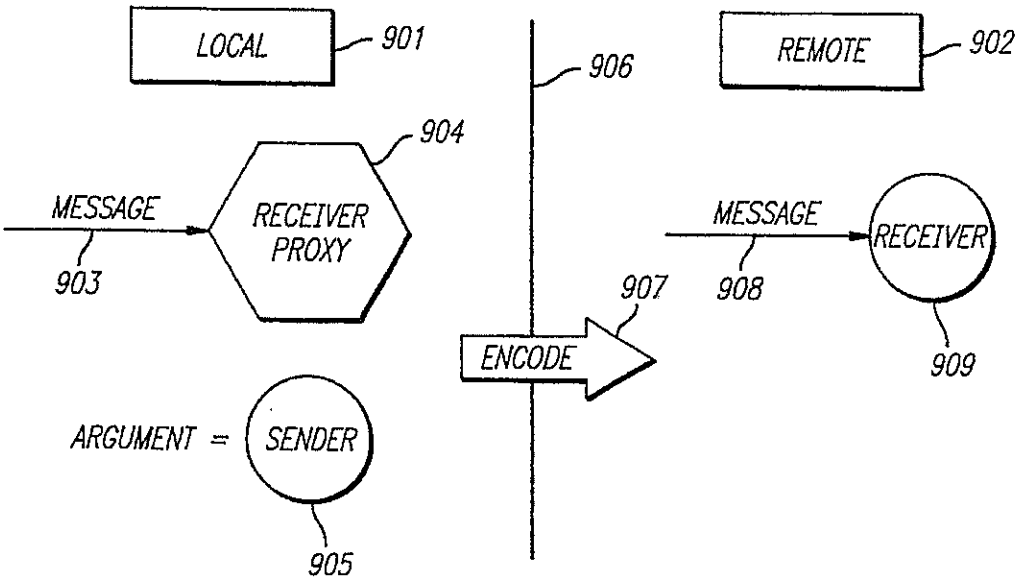



FIG. 3B

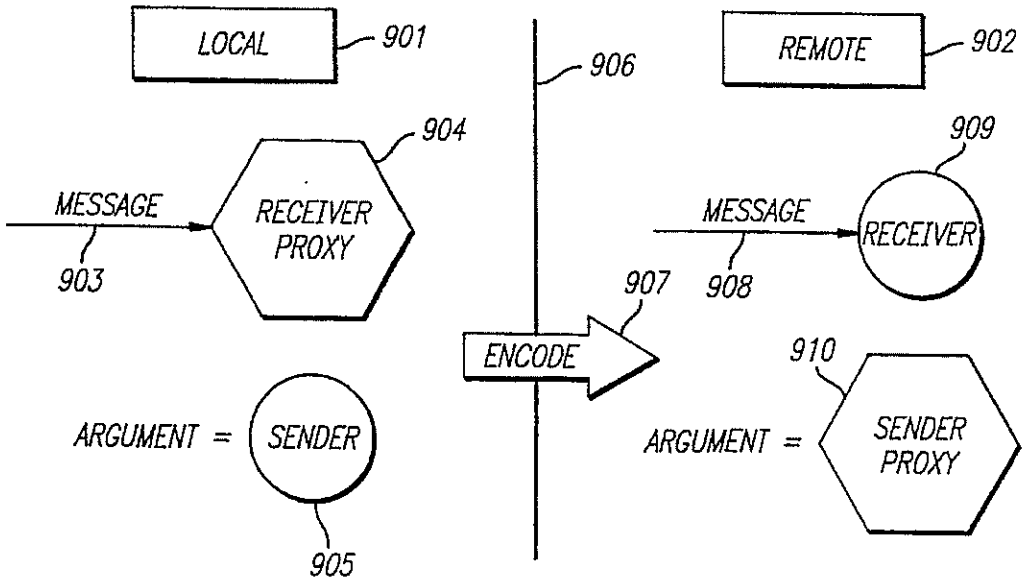


FIG. 3C

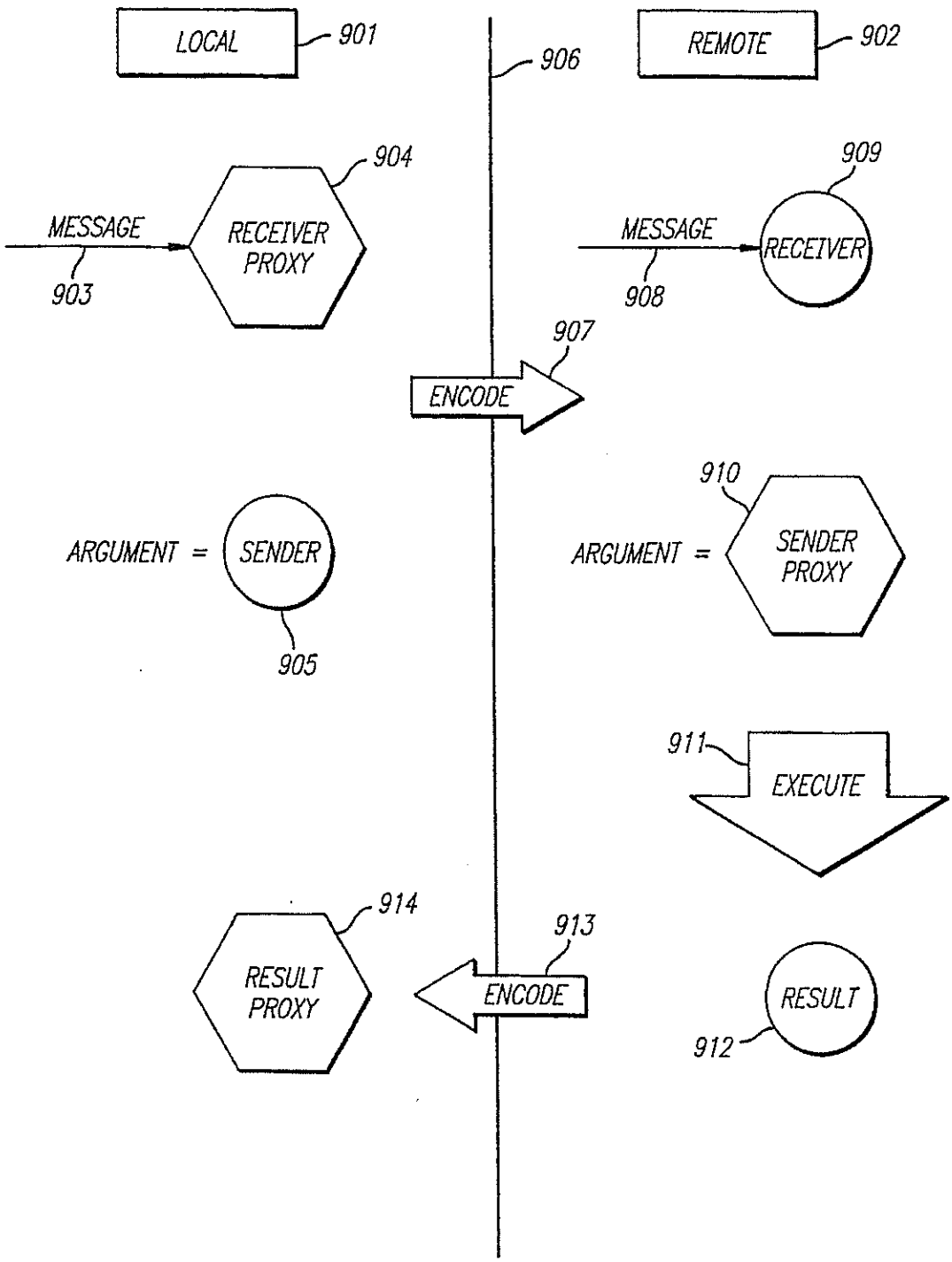


FIG. 5

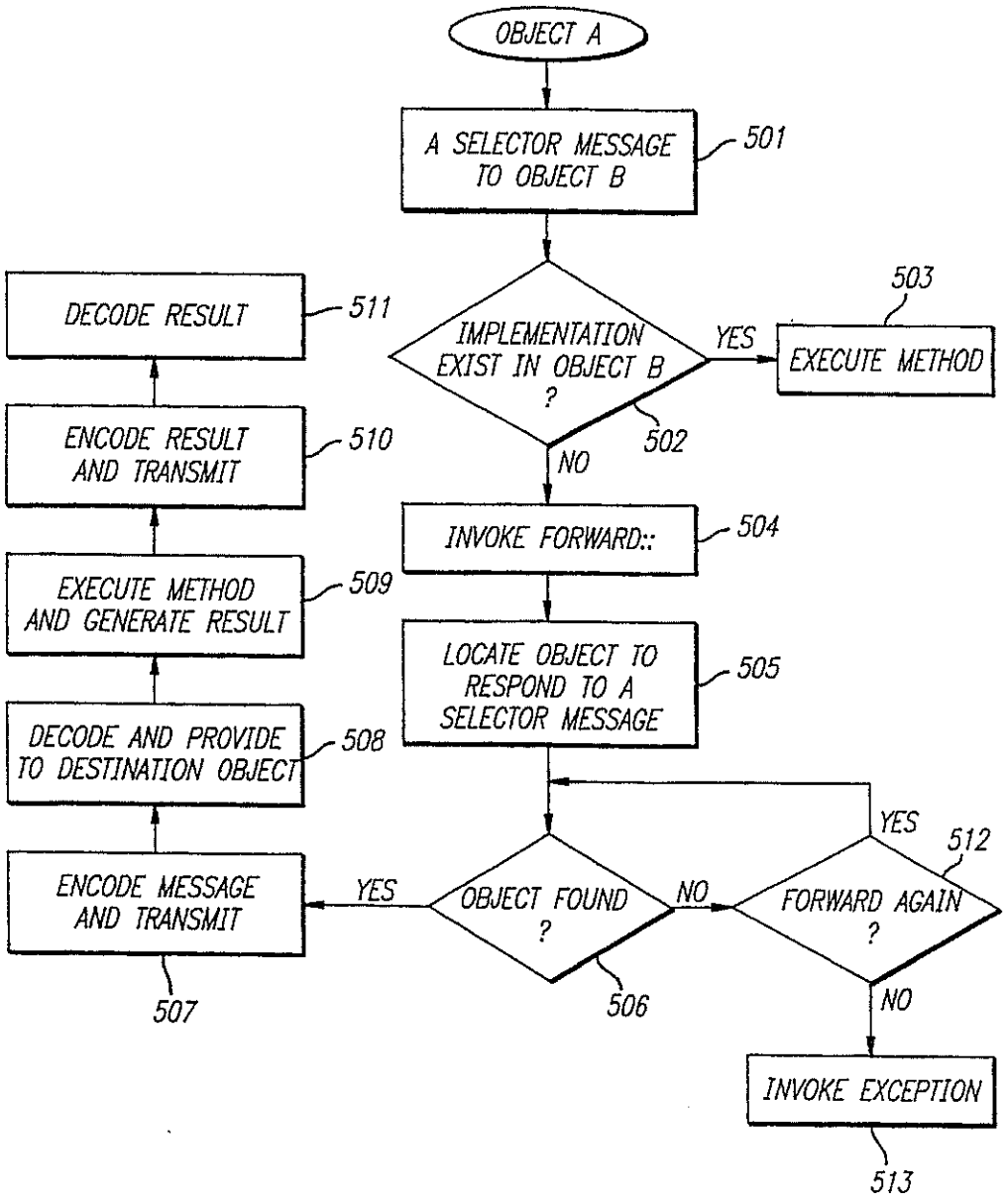


FIG. 6 PRIOR ART

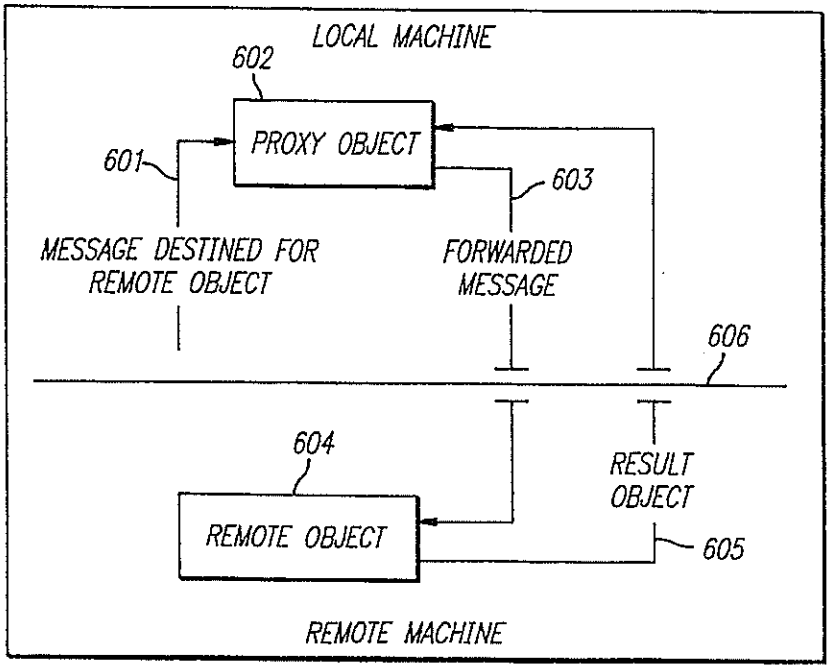



FIG. 7 PRIOR ART

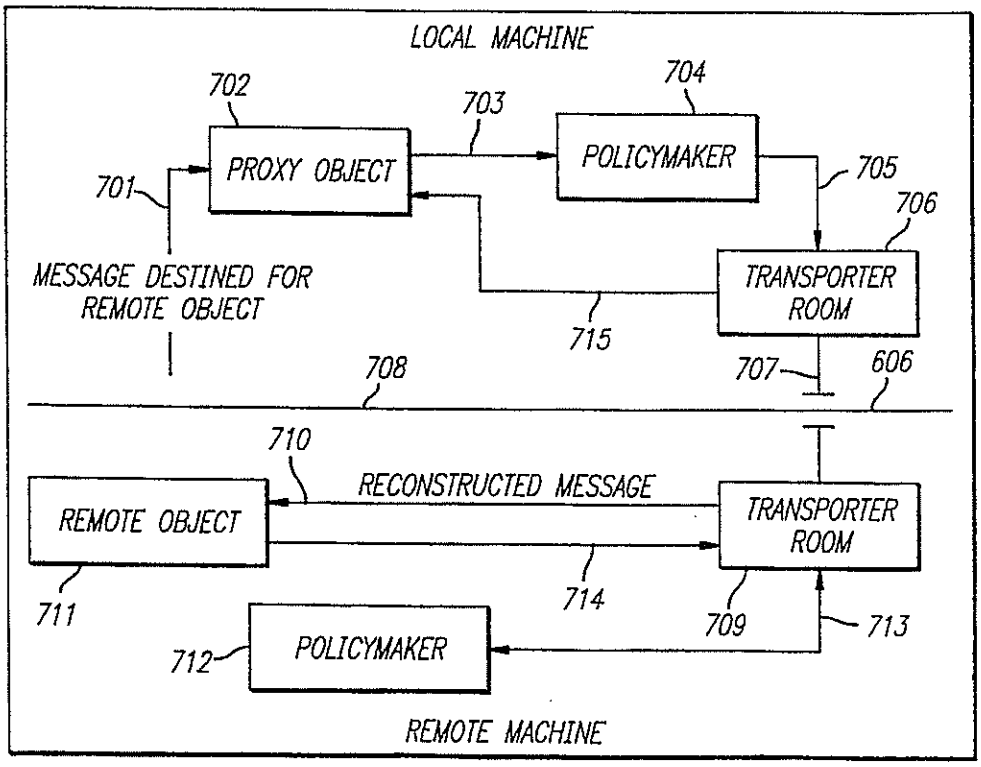


FIG. 8A

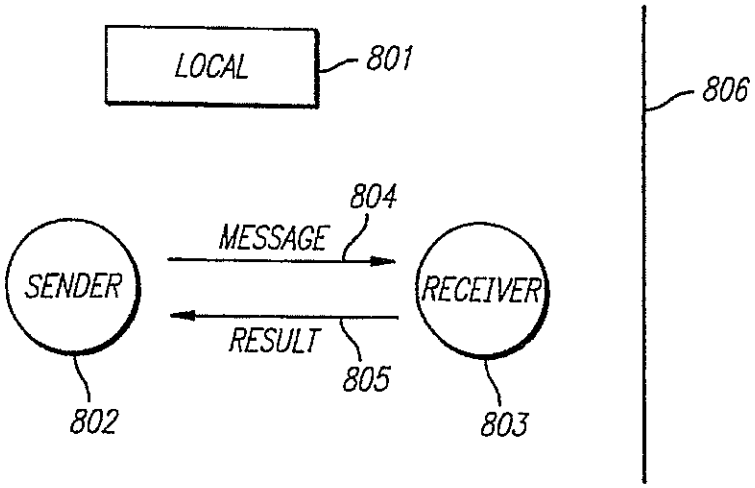
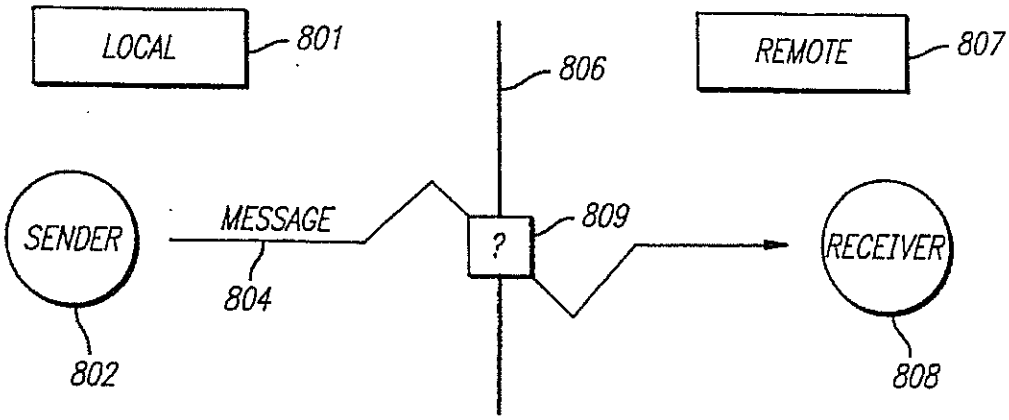



FIG. 8B PRIOR ART



# METHOD FOR PROVIDING AUTOMATIC AND DYNAMIC TRANSLATION OF OBJECT ORIENTED PROGRAMMING LANGUAGE-BASED MESSAGE PASSING INTO OPERATION SYSTEM MESSAGE PASSING USING PROXY OBJECTS

## BACKGROUND OF THE PRESENT INVENTION

This is a continuation of application Ser. No. 07/731,636 filed Jul. 17, 1991, now abandoned.

## FIELD OF THE INVENTION

This invention relates to the field of object-oriented programming and distributed computing.

## BACKGROUND ART

Object-oriented programming is a method of creating computer programs by combining certain fundamental building blocks, and creating relationships among and between the building blocks. The building blocks object-oriented programming systems are called "objects." An object is a programming unit that groups together a data structure (instance variables) and the operations (methods) that can use or affect that data. Thus, an object consists of data and one or more operations or procedures that can be performed on that data. The joining of data and operations into a unitary building block is "encapsulation." In object-oriented programming, operations that can be performed on the data are referred to as "methods."

An object can be instructed to perform one of its methods when it receives a "message." A message is a command or instruction to the object to execute a certain method. It consists of a method selection (name) and arguments that are sent to an object. A message tells the receiving object what to do.

One advantage of object-oriented programming is the way in which methods are invoked. When a message is sent to an object, it is not necessary for the message to instruct the object how to perform a certain method. It is only necessary to request that the object execute the method. This greatly simplifies program development.

Object-oriented programming languages are generally based on one of two schemes for representing general concepts and sharing knowledge. One scheme is known as the "class" scheme. The other scheme is known as the "prototype" scheme. Both the set-based and prototype-based object-oriented programming schemes are generally described in Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems," OOPSLA 86 Proceedings, September 1986, pp. 214–223.

### Class Scheme

An object that describes behavior is called a "class." Objects that acquire a behavior and that have states are called "instances." Thus, in the objective C language, which is the computer language in which the preferred embodiment of the present invention is implemented, a class is a particular type of object. In objective C, any object that is not a class object is said to be an instance of its class. The classes form a "hierarchy." Each subclass in the hierarchy may add to or modify the behavior of the object in question and may also add additional states. Inheritance is a fundamental property of the class scheme and allows objects to acquire



behavior from other objects.

The inheritance hierarchy is the hierarchy of classes defined by the arrangement of superclasses and subclasses. Except for the root classes, every class has a superclass, and any class may have an unlimited number of subclasses. Each class inherits from those classes above it in the hierarchy. Thus, a superclass has the ability to pass its characteristics (methods and instance variables) onto its subclasses.

FIG. 1 is a block diagram that illustrates inheritance. Class 1 (generally indicated by block **101**) defines a class of objects that have three methods in common, namely, A, B and C. An object belonging to a class is referred to as an "instance" of that class. An example of an instance of class 1 is block **102**. An instance such as instance **102** contains all the methods of its parent class. Block **102** contains methods A, B and C.

As discussed, each class may also have subclasses, which also share all the methods of the class. Subclass **1.1** (indicated by block **103**) inherits methods A, B and C and defines an additional method, E. Each subclass can have its own instances, such as, for example, instance **104**. Each instance of a subclass includes all the methods of the subclass. For example, instance **104** includes methods A, B, C and E of subclass **1.1**.

Not all object oriented programming languages permit new methods to be added per instance. For, example, in Objective C, an instance can not have methods that are not contained in its parent class.

A disadvantage of an inheritance-based, object-oriented programming language is object size. Because each subclass must, by definition, include all methods of its parent class and super classes, instances are larger at the bottom of the inheritance hierarchy.

Object-oriented programming languages that utilize the class/instance/inheritance structure described above implement a set-theoretic approach to sharing knowledge. This approach is used in object-oriented programming languages, such as Simula, SmallTalk, Flavors and Loops.

### Prototype Scheme

The prototype scheme is an alternate approach to sharing knowledge in an object-oriented system. In prototype scheme systems that use the individual instances, rather than a class, ("prototypes") are created first, instead of a class. The prototypes are then generalized by defining aspects of their concepts that are permitted to vary. The mechanism for implementing this process is known as "delegation." Examples of prototype languages include the actor language and lisp-based object-oriented systems, such as director, t, and orbit.

Delegation removes the distinction between classes and instances. To create another object that shares knowledge with a prototype, an "extension" object is created that has a list containing its prototypes that may be shared with other objects and containing personal behavior limited to the object itself. When an extension object receives a message, it attempts to respond to the message using the behavior stored in its personal aspect. If the object's personal characteristics are not suitable to answer the message, the object forwards the message on to other prototypes to see if one can respond to the message. This method of forwarding is called "delegating the message."

An example of delegation is illustrated in FIG. 2. Object A provides a message **201** to object B. The message includes a method and arguments to the method. Object B does not

have the method required by the message. Therefore, object B cannot respond to the message. Instead, object B sends the method and message to its delegate. The delegate of object B is object C. Object C has the method requested in the message. The method can then be executed and a response provided to object A.

## Distributed Programming

A disadvantage of current object-oriented programming systems is that all objects are required to exist in a single program or process. This prohibits utilizing an object-oriented programming system when writing distributed applications. In addition, these prior art limitations prevent the creation of applications that are distributed physically over networks of machines.

The difficulty of creating distributed object-oriented programs is illustrated in FIGS. 8A and 8B. In FIG. 8A, all objects are resident in the same program, namely program LOCAL **801**. A sender object **802** sends a message **804** to a receiver **803**. The message may include a method and an argument. The receiver **803** executes the method of the message **804** and returns a result **805**. The result **805** is provided back to the sender **802**. In the example of FIG. 8A, the object-oriented program resides entirely on one side of a boundary **806**.

FIG. 8B illustrates a prior art object-oriented programming system attempting to communicate across a boundary between processes. A local process **801** includes a sender object **802** that generates a message **804** destined for receiver object **808**. However, object **808** is on the opposite side of boundary **806**. That is a separate process identified as REMOTE **807**. An object, such as receiver **808** which resides in a different process than a sender object, is known a "remote object." The language and run time support of the local process **801** does not provide a mechanism to send a message **804** directly to the remote object **808**. At the transition point **809** of boundary **806**, the message is stopped.

One prior art approach for writing distributed applications consists of explicitly defining the boundaries between different programs by specifying protocols, generating client/server stubs, and communicating between processes or programs by using function calls that automatically transport arguments and return values between the client layer and the server layer. At such boundaries, the object-oriented developer can no longer treat items as objects as soon as the boundary of the process is crossed. This defeats the purpose and advantage of using object-oriented programming in the first place.

Another prior art method for providing distributed object oriented programming is described in "Design of a Distributed Object Manager for the SmallTalk-80System," D. Decouchant, OOPSLA 86 Proceedings, September, 1986, pp. 444–452. The Decouchant reference describes the design of a distributed object manager that allows several Small-Talk-80 systems to share objects over a local area network. When a local object desires to communicate with a remote object, the local object communicates with a "proxy" that locally represents the remote object. A proxy is part of the private data of the object manager. The proxy has two fields that describe a remote object, namely, the resident site of the remote object and a virtual pointer to the object in the resident site. If the referenced object migrates, the contents of the referencing object are not modified. The proxy is updated accordingly by the object manager. In this implementation, a proxy is functionally equivalent to a Unix link, except that a proxy is not visible to the programmer. It is a private data structure which is handled by the object manager like other Small-Talk objects.

In the Decouchant reference, three processes cooperate to perform the object manager functions. These are the network manager, the main memory manager and the secondary manager. SmallTalk interpreter processes which access the objects to perform SmallTalk actions may also be present on the site. The network manager is the master process of a SmallTalk site. It ensures consistency of the shared objects with the other network sites and controls the local processes. The main memory manager is in charge of the object management in main memory. It resolves object faults by allocating free space for the missing object and sending an object load request to the secondary storage manager. The secondary memory manager takes care of the object management in secondary storage. This storage is represented by two files, one of which contains the SmallTalk object table and the other one contains the object space.

Another prior art method to provide distributed object-oriented programming is described in "The Design and Implementation of Distributed SmallTalk," John K. Bennett, OOPSLA, Oct. 4–8, 1987, pp. 318–330. SmallTalk itself is a language environment that provides a single user with access to a single object address space. Only rudimentary support exists within SmallTalk for cooperation among users, and no support exists within SmallTalk for object sharing between users or between different machines or between processes on the same or different machines. The Bennett references describes "distributed SmallTalk" (DS) as a method of providing improved communication and interaction among geographically remote SmallTalk users, direct access to remote objects, the ability to construct distributed applications in a SmallTalk environment, and a degree of object sharing among users.

The system described in the Bennett reference does not allow remote classes. Instead, the system requires that classes and instances be co-resident on all processes and machines. This impacts object mobility adversely. Instances can only move to hosts with compatible classes and insuring class compatibility is difficult. In addition, the system of Bennett does not operate in an object-oriented programming system that utilizes class inheritance and reactiveness. (Reactiveness describes the ability of a system to present objects for inspection or modification).

The system of Bennett uses ProxyObjects and a RemoteObjectTable to implement distributed message passing. A ProxyObject represents a remote object to all objects in a local address space. There is one ProxyObject per host per remote object referenced by that host. ProxyObjects cause a remote object's message interface to appear to local objects as if the remote object were locally resident. ProxyObjects redefine the doesNotUnderstand: message of object. This is the primary message defined for ProxyObjects. In other words, messages sent to ProxyObjects are intended to fail. The system responds to this failure by sending the message doesNotUnderstand: to the receiver with the message that was not understood as an argument. The ProxyObject's response to the doesNotUnderstand: message is to forward the original message to the RemoteObjectTable on the appropriate machine or process. The location of the remote object is part of the internal state of the ProxyObject.

The RemoteObjectTable is responsible for receiving and replying to messages forwarded by ProxyObjects. There is one RemoteObjectTable per host. It is the sole instance of

class RemoteObjectTable. The RemoteObjectTable can be thought of as a set of extensions to the object tables (if present) of all remote machines. The RemoteObjectTable keeps track of all local objects that are remotely referenced. When the RemoteObjectTable receives a message from some ProxyObject, it schedules a process that will contain the execution context of the actual message receiver by sending the message perform to the receiver with the forwarded selector and arguments (if any) as arguments to the perform message. The value returned by the perform message is returned to the remote sender in a reply message constructed by the RemoteObjectTable.

Before an object can be sent between processes, the classes must be checked for compatibility. The three cases to consider are:

1. The required class is already present and is compatible;
2. The required class is present, but it is determined to be incompatible; and
3. The required class is not present.

In case 1, the system proceeds normally. In the second case, the attempted move fails and the user is notified of the error. In case 3, the user is asked whether the desired object's class should be moved. If the response is affirmative, the object's super class is checked for compatibility. This procedure continues up the class hierarchy until class object is reached. However, class object may not be moved.

Another method for providing distributed object-oriented programming is described in "Transparent Forwarding: First Steps" Paul L. McCullough, OOPSLA 1987 Proceedings, Oct. 4–8, 1987, pp. 331–341. As in the Bennett system, the McCullough system utilizes ProxyObjects and the doesNotUnderstand: message for identifying and transmitting messages. In the McCullough system, the implementation of doesNotUnderstand: creates an ethernet packet containing the original message and forwards it to the machine containing the remote object. The proxy contains information in its instance variables about where the remote object resides.

FIG. **6** illustrates an overview of the operation of the McCullough system. A message **601** destined for a remote object is provided to a ProxyObject **602**. The ProxyObject instance forms a representation of the message, including both the selector and the arguments, suitable for transmission to a remote machine. This message **603** is forwarded across the process boundary **606** to a remote object **604**. The remote object **604** receives the representation of the message, extracts the message selector and arguments and executes the message send as though it originated on the same machine as the remote object. This result object **605** is transmitted across the process boundary **606** to the ProxyObject **602**. The ProxyObject **602** uses the return representation to reconstruct the result object and returns it to the sender of the original message **601**.

The McCullough system implements four possible message parameter passing schemes, namely, pass by value, pass by reference, pass by proxy and pass by migration. In pass by value, a representation of the object is shipped to the remote machine, which in turn reconstructs the object. Pass by reference cannot be used in a SmallTalk environment because the compiler prevents assignment to formal parameter variables. In pass by proxy, a proxy for the object and any messages which are sent to the proxy are automatically forwarded to the remote object. In pass by migration, we move an object from one machine to another, leaving a proxy object in its prior home.

A centralized control scheme, referred to as PolicyMaker, is used to deliver messages to remote objects. Individual proxies need not record the current network location of a shared object, that is the responsibility of the PolicyMaker. PolicyMaker responsibilities include the decision of whether to pass objects by value, proxy or by migration, and whether to forward a message to a remote object or whether to migrate the object to the local machine for execution. In addition, PolicyMaker keeps track of open connections between machines. For each connection to a remote machine, the PolicyMaker creates an instance of class TransporterRoom. The TransporterRoom takes care of communications protocols between machines, as well as the linearization of messages and objects.

FIG. **7** illustrates the flow control of sending a message from a machine to a remote object using the scheme of the McCullough system. A message **701**, destined for a remote object, is provided to a ProxyObject **702**. The sender of the message **701** believes it is sending to a local object, but in reality it is sending to a remote object. The ProxyObject **702** sends a message **703** to the local PolicyMaker **704**. The PolicyMaker **704** determines whether the arguments of the message should be sent by copying or by proxy to the remote object. The PolicyMaker establishes a connection to the remote machine via transporter room **706**. The PolicyMaker **704** provides the message **705** to the TransporterRoom **706**. The TransporterRoom **706** linearizes and transmits the message as message **707** to the remote machine across process boundary **708**.

The TransporterRoom **709** of the remote machine receives the message **707**. The TransporterRoom **709** sends the reconstructed message **710** to the remote object **711**. The remote object **711** returns a message **714** to the TransporterRoom **709**. The PolicyMaker **712** considers the resulting object and determines whether to return it by value or by proxy and communicates to the TransporterRoom **709** on path **713**. The TransporterRoom **709** sends the message **707** to the TransporterRoom **706** across process boundary **708**. The TransporterRoom **706** reconstructs the result object and provides it as message **715** to proxy object **702**, which can then return it to the sending context.

The use of migration limits the performance and ease of use of these prior art schemes. Migration of objects from their home process adds to the complexity of the system. Another disadvantage of these prior art schemes is that each process and thread must be forked to anticipate each expected iteration. There is no provision for dynamic recursive communication between processes. In addition, these prior art schemes rely on a pure, large object oriented language/environment, such as SmallTalk. This requires substantial run time support to implement communication between processes. In addition, the prior art schemes do not implement suitable object collection methods.

## SUMMARY OF THE INVENTION

The present invention permits the distribution of objects and sending of messages between objects that are located in different processes. Initially, a "proxy" object is created in the same process as a sender object. This proxy acts as a local receiver for all objects in the local program. When the proxy receives a message, the message is encoded and transmitted between programs as a stream of bytes. In the remote process, the message is decoded and executed as if the sender was remote. The result follows the same path, encoded, transmitted, and then decoded back in the local process. The result is then provided to the sending object.

## BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram illustrating the concept of inheritance in object-oriented programming.

FIG. 2 is a block diagram illustrating delegation in object-oriented programming.

FIGS. 3A–3C illustrate the distributed processing object oriented programming system of the present invention.

FIG. 4 is a block diagram illustrating a general purpose computer system for implementing the present invention.

FIG. 5 is a flow diagram of the forwarding method of the present invention.

FIG. 6 is a block diagram illustrating the prior art distributed processing object-oriented programming system.

FIG. 7 is a block diagram illustrating another prior art distributed processing object-oriented programming system.

FIGS. 8A and 8B illustrate non-distributed programming systems.

## DETAILED DESCRIPTION OF THE INVENTION

A method and apparatus for distributed execution of methods is described. In the following description, numerous specific details, such as object-oriented programming language, operating system, etc., are set forth in detail in order to provide a more thorough understanding of the present invention. It will be apparent, however, to one skilled in the art, that the present invention may be practiced without these specific details. In other instances, well known features have not been described in detail so as not to obscure the present invention.

The present invention may be implemented on any conventional or general purpose computer system. An example of one embodiment of a computer system for implementing this invention is illustrated in FIG. 4. A keyboard **410** and mouse **411** are coupled to a bi-directional system **419**. The keyboard and mouse are for introducing user input to the computer system and communicating that user input to CPU **413**. The computer system of FIG. 4 also includes a video memory **414**, main memory **415** and mass storage **412**, all coupled to bi-directional system bus **419** along with keyboard **410**, mouse **411** and CPU **413**. The mass storage **412** may include both fixed and removable media, such as magnetic, optical or magnetic optical storage systems or any other available mass storage technology. The mass storage may be shared on a network, or it may be dedicated mass storage. Bus **419** may contain, for example, 32 address lines for addressing video memory **414** or main memory **415**. The system bus **419** also includes, for example, a 32-bit data bus for transferring data between and among the components, such as CPU **413**, main memory **415**, video memory **414** and mass storage **412**. Alternatively, multiplex data/address lines may be used instead of separate data and address lines.

In the preferred embodiment of this invention, the CPU **413** is a 32-bit microprocessor manufactured by Motorola, such as the 68030 or 68040. However, any other suitable microprocessor or microcomputer may be utilized. The Motorola microprocessor and its instruction set, bus structure and control lines are described in MC68030 User's Manual, and MC68040 User's Manual, published by Motorola Inc. of Phoenix, Ariz.

Main memory **415** is comprised of dynamic random access memory (DRAM) and in the preferred embodiment of this invention, comprises 8 megabytes of memory. More or less memory may be used without departing from the scope of this invention. Video memory **414** is a dual-ported video random access memory, and this invention consists, for example, of 256 kbytes of memory. However, more or less video memory may be provided as well.

One port of the video memory **414** is coupled to video multiplexer and shifter **416**, which in turn is coupled to video amplifier **417**. The video amplifier **417** is used to drive the cathode ray tube (CRT) raster monitor **418**. Video multiplexing shifter circuitry **416** and video amplifier **417** are well known in the art and may be implemented by any suitable means. This circuitry converts pixel data stored in video memory **414** to a raster signal suitable for use by monitor **418**. Monitor **418** is a type of monitor suitable for displaying graphic images, and in the preferred embodiment of this invention, has a resolution of approximately 1020× 832. Other resolution monitors may be utilized in this invention.

The computer system described above is for purposes of example only. The present invention may be implemented in any type of computer system or programming or processing environment.

The preferred embodiment of the present invention implements an object-oriented programming system using objective C language. Objective C is an extension to ANSI C that supports the definition of classes of objects and provides syntactic and run-time support for sending messages to objects. This language model is partially derived from SmallTalk and has been described in "Object-Oriented Programming; An Evolutionary Approach," Brad J. Cox, Addison-Wesley 1986 and in "SmallTalk-80: The Language and its Implementation," Adele Goldberg, Dave Robson, Addison-Wesley 1983.

One feature of objective C is "dynamic binding" of messages to the actual methods to be invoked, depending on the class of the receiver. A programmer writing code in objective C can create code that sends a message "doSomething" to an object. The actual method corresponding to the class of the target object does not need to be determined until the message must be sent. This allows objects of any classes that implementing the doSomething method to be substituted for the target object at run time without having to modify the part of the program that sends the message. Also, in objective C, programs have run time access to method "signatures," that encode a method's argument and return types for each class. The method signature provides a way for two programs to agree on the format of messages. Moreover, there is a way to extract arguments from the stack using the signature.

In its preferred embodiment, the present invention is implemented in a computer system using an object-oriented operating system. One such object-oriented operating system is known as the "Mach" operating system and is implemented on computers manufactured by NeXT, Inc., the Assignee of the present invention.

The Mach operating system is an object-oriented operating system that supports distributed programming. It is a multi-tasking operating system kernel, allowing multiple, independent "tasks," which provide the basic environment of a program in the form of a demand-paged virtual "address space" and one or more "threads" of execution. Mach supports message-passing within and between tasks. This support is distinct from objective C messaging described earlier.

Fundamental to Mach's ability to deliver messages between different programs is an abstraction called a "port." A Mach port is a buffered communication channel over which messages are sent. This channel, which is maintained by the operating system, may be local, may span two tasks on the same machine, or may span tasks on different machines. The physical location of the receiving end of a

port has no effect on the sender, which always sees a local reference to the port.

The messages sent on a port are buffered, that is, a sender writes messages to a port with a "send" primitive, and a receiver accepts messages with a "receive" primitive. The messages themselves may be of any size, and consist of a header followed by zero or more data objects. For efficiency, large array arguments are passed out-of-line with copy-on-write semantics. Of particular interest is the ability to pass Mach ports themselves as data objects in a message. By this means, one task may pass a port to another, with the kernel maintaining address translation along the way. This allows tasks to learn about the existence of new external objects, or make new "acquaintances," by receiving their ports in a message. Thus, ports may also be viewed as a reference for an object that is independent of any particular space, and may be freely passed between programs.

For a task to communicate with a "receiver" object in another address space, it must first establish a connection with that program, and then create a local "proxy" for the object. When a message is sent to this proxy, the elements of the objective C message are encoded into a Mach message, which is then forwarded through a Mach port to the other program. On the receiving side, the message is received, decoded, and then forwarded it to the target objective C object. The return value of the objective C method is then encoded and sent back to the originator, where it is decoded and returned as the value. Each part of this model is now described.

In order to communicate with an object in a different program or process, that program or process must be known. The present invention uses Mach ports to represent "domains" of objects, and a token to identify objects within that domain. This two-part address is easily communicated, since the port maintains its identity as it moves between domains. In Mach, acquiring ports is synonymous with acquiring privileges to communicate. The present invention requires that the local domain has send rights to a port that the remote domain has receive rights to, and also that the remote domain has send rights to a port that the local domain has receive rights to, before any communication can take place. Thus, mutual consent is required to communicate.

In addition to learning of each other's ports, each domain must also provide the other with the token corresponding to its "first proxy." A first proxy is required to bootstrap the communication. There must be at least one known object in a remote domain before a message can be sent to it. Because a connection allows messages in either direction and initiated by either party, each side of a connection must have a first proxy. This first proxy may be viewed as the "receptionist" for the remote domain, as other objects are obtained (discovered) by asking this object. This object also is a candidate for implementing sender authentication.

The present invention provides a means for implementing an extensible, distributed program in which one task is responsible for creating other tasks to communicate with. This is a master/slave relationship; the master can provide the slave with send rights to the master's port as part of the creation process. When the slave starts executing, it sends a Mach message containing send rights to its port and a token for its first proxy back to the master. The master then replies with an indication of whether the connection is granted, and what token to use for the first proxy. This "bootstrap-meta-protocol" results in both tasks knowing about each other, allowing communication to ensue.

## Distributed Object Oriented Programming

The present invention provides a method for different processes to communicate, using a traditional language-based, message-passing paradigm. The present invention has a number of advantages over prior art methods of distributed object-oriented programming. These advantages include no pre-defined set of messages, transparent to the programmer, no code generation step and a method for bridging the gap between object-oriented languages and object-oriented operating systems.

The present invention differs from the prior art approaches of Decouchant, Bennett and McCullough. The present invention uses an object-oriented superset of ANSI C with minimal run time support to implement transparent messaging between application programs as opposed to the prior art systems, that rely on a pure, large, dynamic object-oriented language/environment, such as SmallTalk. The present invention either implements a client/server setup or a master/slave setup that involves forking tasks to perform background operations and using the present invention to communicate with these tasks. Because communications are serialized by the operating system, remote messages performed by a slave task appear to the program just like other asynchronous events, such as mouse clicks, allowing the design of a consistent user interface. Modularity, extensibility and safety are gained by spawning new tasks. The ability to implement recursive remote messaging is an important feature of the present invention, for example when user interaction is required to perform the desired task.

The present invention can also be implemented in a client/server setup. In the client/server setup, both the client and the server begin independently. Communication is through an agreed upon method. The client and server can communicate with each other by looking for a manager for the communications channel (e.g. network).

The present invention provides a new alternative for developing extensible programs. Instead of adding functionality by adding code that defines new object classes and then loading that code into a main program, a new program is created and forked. If there are any errors in the new program, they reside outside the main program improving performance. In addition, processing is parallel and asynchronous, leading to improved performance.

In one embodiment, the first time a message is sent to a proxy, the receiver is asked for the method signature in order to allow the proxy's domain to encode the arguments. This can increase communication time. One alternative embodiment provides for prior agreement between processes so that two tasks know in advance the method signatures for their proxies. Alternatively, when prior agreement is not possible due to the dynamic nature of the required exchange, the atomicity of the signature request can be changed to communicate all the public signatures for a given proxy, resulting in a single "meta-protocol" transaction for the proxy.

When an object is passed by reference, a new proxy is typically created. The present invention includes safeguards to guarantee that if a remote object is encoded twice, the same proxy (in the pointer equality sense) will be obtained in the local domain. This unicity of proxies is maintained by a table which maps tokens of remote objects to their local proxies and looks for previously created proxies when an object pass by reference is decoded. The present invention keeps all proxies until the communication with the remote domain ends. At that time, the table is used to de-allocate all proxies.

The operation of the present invention is illustrated in FIGS. 3A–3C. Referring first to FIG. 3 A, a local process **901** is separated from a remote process **902** by boundary **906**. The boundary **906** could be a separation between

programs on the same computer or it could represent the separation of two different machines on a network. The local process **901** includes a sender object **905** that sends a message to receiver object **909**. The object **909** is located in the remote process **902**. However, the sender object **905** can send its message **903** to the remote object as if it were a local object.

The local process **901** includes a receiver proxy **904** that accepts the message **903**. The receiver proxy **904** is an object that executes a forward:: method. The receiver proxy **904** encodes the message and transmits it across the process boundary to the remote object. In the preferred embodiment of the present invention, the proxy **904** encodes the message, (which is a language based message such as, for example, an objective C message), as an operating system message, such as a Mach message **907**, and transmits it to the receiver object **909** in the remote process **902**. The receiver object **909** decodes the Mach message into a language based message for execution or handling in the remote process **902**.

The present invention supports nested, recursive, remote messages. That is, when a message is sent to a remote object, that remote object may send other messages back to the local process (which may again send other remote messages), as part of its calculations before providing a reply to the initial message. These messages may be nested arbitrarily deep. If only the sender itself is sent as an argument, (such as illustrated in FIG. **3** A), the receiver determines what further information is required from the sender and sends messages back to the local process to obtain that information before generating a reply.

Referring to FIG. **3** B, the receiver object **909** requires additional information from the sender object so it generates a request message to the sender object **904**. However, since the sender object **904** is not resident in the remote process **902**, a sender proxy **910** is created in the remote process **902**. The sender proxy **910** encodes the request into a Mach message and transmits it across process boundary **906** to sender object **905**. The Mach message is decoded into a language based message and generates a response. This response is sent back to the receiver object **909**, (again via receiver proxy **904**).

Referring to FIG. **3** C, the receiver object **909** then performs an execute **911** on the original message to generate a result object **912**. The result object **912** is encoded **913** and transmitted across process boundary **906** to result proxy **914** in local process **901**.

In the present invention, proxies are not required to be created in advance. Once one proxy of a remote process exists, N proxies can be created for communication with that process. The present invention also permits the sending of objects themselves across process boundaries.

The recursive nature of the present invention is useful when the remote object does not recognize a method sent to it by a local object. For example, the local object may send a message to a remote object requesting it to execute the method "foo". If the remote object does not recognize the method, it can ask the sending object "what is foo?" All objects recognize the method "what is". The sender object can then respond with instructions concerning the nature of the method being investigated. For example, the sender can reply that foo is a method that requires an integer. If no integer has been provided, the remote object can then request the integer.

The creation of the first proxy is provided automatically in the present invention. The first time a process is accessed, it must call a process referred to as the "proxy receptionist". By definition, the first call to a new process is to be at sequence number 0. The first proxy by definition has a sequence number of 0. Subsequent proxies, as needed, are defined and the sequence numbers are provided to the other process.

In the preferred embodiment of the present invention, communication between processes is implemented in a master/slave or client/server relationship. In the case of a client/server relationship, the server may "publish" its port in an appropriate place on the system. The client looks up this port and then uses it to initiate the bootstrap-meta-protocol described above.

The establishment of a connection defines only the first proxy on each side. Proxies for any other objects that need to be communicated are created dynamically as they are encountered. For example, if a remote method returns a new object, a new proxy is created when decoding the result locally, such that the local program could send a message to this new remote object without any explicit setup.

Once a connection is established, a message may be sent (in either direction). To send a message, the arguments must be encoded into a form that is representable in a Mach message, so that the receiver may decode them correctly. To do this, the sender must know the method signature of the message. The method signature is part of the "protocol" that both sender and receiver must understand in order to communicate, and is a domain-independent encapsulation of the method name, its argument types, and its return value type.

Although a protocol may be arranged by prior agreement in many cases, the present invention determines the protocol dynamically by asking the receiver how to encode the arguments for each message as it is encountered. During the first attempt to send a given message to a remote object, the local domain consults the remote domain to get the signature corresponding to the actual class implementation that will ultimately receive the message. This method signature is cached on a per-connection basis, with the assumption that the class of a remote object will not change for the duration of the connection. Thus, subsequent messages use the cached signature, and do not have the overhead of this "meta-protocol" transaction. This dynamic aspect is useful for type checking, and allows one program to "learn" how to talk to another program.

The argument encoding for standard C data types is explicit and strictly pass-by-value, and maps substantially directly onto a Mach message. Pointers to C data structures are not encoded. Arguments that are first-class objects (as opposed to simple C data types) are handled specially; they are asked to encode themselves. The default encoding scheme that most objects inherit is to allocate a token (if one does not already exist) in the local domain, and encode only that token. Along with the port of the local domain, this constitutes an object reference (via proxy), and when it is decoded on the receiving side, results in a new proxy for that remote object.

Thus, first-class objects by default get passed by reference instead of by value. For example, when domain A sends a message with object X as an argument to a remote object in domain B, a new proxy is created in domain B to represent X. Any subsequent message that domain B sends to X results in another remote object transaction. In the present invention, an implementation of an object class is free to choose to implement a different encoding scheme, for example one that encodes the object by value, though this requires that both sender and receiver implement that class of object. The

"pass-by-proxy" scheme does not have this restriction and is generally preferred.

Because objective C implements functional messages, a return value is always returned in a reply message. It is encoded exactly the same way as arguments are. In particular, if the result is an object or a proxy, it is encoded as a proxy or an object, respectively, in the other domain. The reply message also encodes information about errors or exceptions that may have occurred, so that they can be raised in the local context. That is, an error occurring while executing a message sent to a remote object is caught and returned to the caller, so that the exception is raised in the local domain. This makes error handling transparent. Remote exceptions may be handled the same way as local exceptions.

The use of a single port to represent a domain of objects allows an efficient and simple implementation. A one-way message is used when forwarding a message to a remote domain. Messages are received on the local domain's single published port while waiting for the reply. Since all messages (including both the reply message and any other messages initiated from remote domains) arrive on the same port, each one can be handled serially. There is no global state to keep track of (the logic is implemented in a reentrant manner), and each message and reply have matching sequence numbers, so it can be determined when the correct reply has been received.

In effect, the low level routine to actually forward the message to the remote domain becomes the main loop of the program until the reply is received. There may be many nested levels of this low level routine at one time. Outside the scope of a locally-initiated remote message (i.e., the "idle" state of waiting asynchronous messages to arrive), the local domain's port is listened to by the main program, along with other non-remote-object related events. This approach contrasts with that of the prior art McCullough system, where a process or thread is forked to field every expected reply. The present invention achieves an order of magnitude in performance by avoiding the forking when communicating.

An example of a computer program listing that may be used to implement the present invention is described in Appendix A. This computer program listing is given by way of example only. The present invention may be practiced using other programs and methods as well.

## Automatic Forwarding of Messages

The present invention, in its preferred embodiment, takes advantage of a method referred to as "automatic forwarding of messages." This method is the subject of copending patent application Ser. No. 07/695,316 filed May 3, 1991, entitled "METHOD FOR PROVIDING AUTOMATIC FORWARDING OF MESSAGES AND METHODS" and assigned to the assignee of the present invention. This method is described below.

In objective C, when an object receives a message that contains a method that the object does not recognize, an exception is provoked leading to an error. The present invention, instead of provoking an exception, redirects the message to an acquaintance that can understand the message. For example, if an object receives a message containing a method that the receiving object does not contain, the message is forwarded to an acquaintance object that does contain the method. This provides the advantage of inheriting the method from the acquaintance object but does not require the first receiving object to actually have the method itself. This reduces code size the memory requirements.

The present invention has a plurality of uses. For example, a new object class can be defined so that one of its instances (attributed object) adds an attribute to another object (its forwardee). In that situation, automatic forwarding occurs when an attributed object receives a message that is irrelevant to the attribute, and the method is forwarded to the forwardee. In another situation, some functionality is applied before and/or after the forwarding. This can be used in the case of a locking data structure where all methods must be redirected to the locked data after acquiring the lock, and where the lock must be released after the execution of the forwarded method. The present invention also has use in the case of forwarding messages in a distributed environment where there is no explicit forwardee. Rather, there is a network address of the forwardee.

The implementation of the present invention in the preferred embodiment requires trapping the "message not recognized" exception of objective C, retrieving all of the arguments of the unrecognized message, and sending the "forward::" message to the object.

The resulting automatic forwarding system of the present invention is more powerful than multiple inheritance and is transparent to the programmer and developer. The system is general, because the forwardee is not explicit, thus permitting solutions to a large class of problems.

The present invention uses the forward:: command so that subclasses can forward messages to other objects. The format is forward: (SEL) aSelector:(marg_list) argFrame. When an object is sent an aSelector message, and the run time system cannot find an implementation of the method for the receiving object, the run time system sends the object a forward:: message to give it an opportunity to delegate the message to another object. If the forwardee object cannot respond to the message either, it also has the opportunity to forward the message. A forward:: message is generated only if a selector method is not implemented by the receiving object's class or by any of the classes it inherits from.

The forward:: message thus allows an object to establish relationships with other objects that will, for certain messages, act on its behalf. The forwarding object is, in a sense, able to "inherit" some of the characteristics of the object it forwards the message to. The forwarding object is not limited to the forwardees it may select, and a forwardee relationships may be formed with more than one object at the same hierarchical level. Therefore, the present invention provides the advantages of multiple inheritance without the code size problem.

In addition to forwarding messages, the forward:: method can locate code that responds to a variety of different messages, thus avoiding the necessity of having to write a separate method for each selector.

If implemented to forward messages, a forward:: method has two tasks. First, to locate an object that can respond to the aSelector message (this need not be the same object for all messages). Second, to send the message to that object using the performv:: and performv method.

The operation of the present invention is illustrated in the flow diagrams of FIG. 5. At step **501**, Object A sends an aSelector message to object B. At decision block **502**, the argument "Implementation exists in Object B?" is made. If the argument is true, the method of the aSelector message can be executed by object B. If that is the case, the system proceeds to step **503** and the method is executed. If the argument is not true, the system proceeds to step **504** and invokes the forward:: method.

The forward:: method then performs the first of its two tasks at step **505**. Namely, it attempts to locate an object to respond to the aSelector message. At decision block **506**, the argument "Object found?" is made. If the argument is true, then forward:: has successfully found an object to respond to the aSelector message. In the present invention, the forwarding object is typically a proxy object.

The system then proceeds to step **507**, the message is encoded and transmitted as an operating system message to another process. At step **508**, the operating system message is decoded and provided to the destination object. At step **509**, the destination object executes the method of the message to generate a result. At step **510**, the result is encoded and transmitted to the first process as an operating system message. At step **511**, the message is decoded and the result is provided to the sending object.

If the argument at decision block **506** is not true, the system proceeds to decision block **512**. At decision block **512**, the argument "forward again?" is made. If the argument is true, the message is forwarded again and a search for an object to respond to the message is made. If the argument is false, the system proceeds to step **513** and an exception (error) is invoked.

In the case in which an object forwards messages to just one destination, a forward:: method could appear as follows:

```
- forward: (SEL)aSelector :(marg_list)argFrame
{
    if ([friend respondsTo:aSelector])
            return [friend performv:aSelector:argFrame];
        return [self doesNotRecognize:aSelector];
}
```

ArgFrame is a pointer to the arguments included in the original aSelector message. It is passed directly to performv:: without change. The default version of forward:: implemented in the object class invokes the does not recognize: method. It does not forward messages. Thus, if a user chooses not to implement forward:: methods, unrecognized messages will be handled in the usual way.

The objective C run time code routines for implementing automatic forwarding of messages is as follows:

```
// provide a default error handler for unrecognized messages
static id-forward (id self, SEL sel, . . .)
{
    id      retval;
    // the following test is not necessary for Objects (instances of
    Object)
    // because forward:: is recognized.
    if (sel ==@selector (forward::)) }
            _objc_error (self, _errDoesntRecognize, SELNAME
    (sel));
        return nil;
    }
    retval =[self forward: sel : &self];
    return retval;
    {
        Method smt =(Method) objc_malloc (sizeof (struct
        objc_method));
        smt->method_name sel;
        smt->method_types = "";
        smt->method_imp = (IMP)_forward;
        _cache_fill (savCls, smt);
    }
    {
        Method smt = (Method) objc_malloc (sizeof (struct
        objc_method));
        smt->method_name = sel;
        smt->method_types = "";
        smt->method_imp = (IMP)_forward;
            cache_fill (savCls, smt);
    }
    // the class does not respond to forward: (or, did not supply a
    dest)
    {
        Method smt = (Method) objc_malloc (sizeof (struct
        objc_method));
        smt->method_name = sel;
        smt->method_types = "";
        smt->method_imp = (IMP)_forward;
            _cache_fill (savCls, smt);
    }
    return (IMP)_forward;
}
```

Thus, a method and apparatus for providing distributed processes is described.

APPENDIX A

```
#import <stdlib.h>
#import <stdarg.h>
#import <objc/HashTable.h>
#import <objc/hashtable.h>
#import <sys/message.h>

/* Declarations that will go away */
extern SEL _sel_registerName(STR key);

/*****************       Definitions      ***************************/

extern int defaultCommTimeout; /* in millisecs, <0 means infinite */

extern int remoteMessageReceiveCount; /* increments when a msg is received */

/*      The following type of function may be passed to the beginListeningOn:rootObject: m
ethod. It gets asynchronously called during remote message sending. For example, an app co
uld register the port and function with DPSAddPort when the boolean is YES (and DPSRemoveP
ort when NO). */
typedef void (*remote_message_handler_t)(msg_header_t *msg, void *userData);

typedef void (*receive_enable_proc_t)(port_t port, remote_message_handler_t fun, BOOL shou
ldEnable);

typedef enum {
    #define REMOTE_EXCEPTION_BASE 36000 /* less than appkit base */
    /*  Format of exceptions is a label and a message string    */
    GENERIC_REMOTE_EXCEPTION = REMOTE_EXCEPTION_BASE,
    TIMEOUT_REMOTE_EXCEPTION,
    LAST_REMOTE_EXCEPTION
} RemoteException;

/*****************       Communication    ***************************/

@interface Communication: Object {
    @public
    port_t      sendPort;
    int         timeout;        /* in milliseconds */
    NXHashTable *objectsGivenAway;
    NXHashTable *allProxies;
}
+ beginListeningOn:(port_t)listenPort enableProc:(receive_enable_proc_t)aProc;
    /* Initialize the remote object system.
    enableProc is called immediately with a boolean of YES. */

+ new:(port_t)port timeout:(int)aTimeout;
+ findCommForPort:(port_t)aPort;

@end

/*****************       Remote Objects  ***************************/

@interface RemoteObject: Object {
    Communication       *comm;  /* nil means "local" */
    unsigned            name;           /* object name; 0 means localRoot */
    HashTable           *knownSelectors;        /* cache */
}

+ messageReceived:(msg_header_t *)msg;
```

```
+ newRemote:(unsigned)remoteName withCommunication:(Communication *)communication;
    /* This is used only for bootstrap */

+ registerLocalRoot:root;
    /* Register the local root, and return a remote object with name 0;
    Can only be called once */

+ newLocal:local withCommunication:(Communication *)communication;
    /* search for a local-RemoteObject that corresponds to local id.
    If none found, creates one */

- (unsigned)remoteObjectName;

- (unsigned) methodArgSize: (SEL) sel;
    /* Size of the arguments of the remote object, including self and sel;
    0 iff error */
- forward: (SEL) sel : (void *) args;
@end

/*****************       Encoding Protocol        **************************/

@interface Object (Object_MakeRemote)
- encodeRemotelyFor:(Communication *)communication freeAfterEncoding:(BOOL *)flag;
    /* This method is called for each object being encoded; By default, it consists in cre
ating a new "local" remote object (i.e. passing the object by reference).  To pass the obj
ect by value, just return the object.  To substitute another object, just return it.
    Iff flag is set, the returned object will be freed after encoding. */

- afterPortReading:(Communication *)communication;
    /* This method is called after decoding an object to give an opportunity to replace it
; original object can be freed */

- instantiateObject:(const char *)className;
- setOutlet:(const char *)outletName with:dest;
@end
```

```
#import "RemoteObject.h"
#import "NXPortStream.h"

#import <string.h>
#import <stdio.h>
#import <libc.h>
#import <cthreads.h>
#import <mach.h>
#import <syslog.h>
#import <objc/error.h>
#import <objc/List.h>
#import <objc/objc-runtime.h>
#import <kern/mach_param.h>
#import <sys/message.h>

/*****************      Forward Definitions      ***************************/

static void handleRemoteMessage(msg_header_t *m, void *userData);
static void remoteAsk (port_t port, msg_header_t *msg, int timeout);
#if 0 // Never tried
static void remoteTell(port_t target, msg_header_t *msg, port_t sender, int timeout);
#endif

/*****************      Utilities      ***************************/

int defaultCommTimeout = 15000;
BOOL enableWarning = NO;

static void logv(const char *format, va_list args) {
    printf("RemoteObjects[pid %d]:\t", getpid());
    vprintf(format, args);
}

static void ROLog(const char *format, ...) {
    va_list     args;
    va_start(args, format);
    logv(format, args);
    va_end(args);
}

static void warning(const char *format, ...) {
    va_list     args;
    va_start(args, format);
    if (enableWarning) logv(format, args);
    va_end(args);
}

static void error(const char *format, ...) {
    va_list     args;
    va_start(args, format);
    logv(format, args);
    va_end(args);
    NX_RAISE(GENERIC_REMOTE_EXCEPTION, "Internal error", NULL);
}

static int generateSequenceNumber() {
    static int number=0;
    number++;
    if (!number) number = 1;    /* useless */
```

```
    return number;
}

@interface Object (Private_Imports)
- reallyFree;
@end

/*****************      Selector Info   *************************/

@interface RemoteMethodInfo: Object {
    NXAtom      typedesc;
}

static RemoteMethodInfo *knownRemoteMethodInfo = nil;

+ localMethodInfoFor:(Class)class :(SEL)sel;
    /* Return RemoteMethodInfo for a local method;
    Can return nil */

- encodeMethodParams:(void *)args onto:(NXPortStream *)stream;
    /* encode the method frame onto stream (excluding self and sel) */

- (void *)decodeMethodParamsFrom:(NXPortStream *)stream;
    /* decode the method frame from stream;
    return a freshly malloced pointer, never NULL (unless error) */

- encodeMethodRet:result onto:(NXPortStream *)stream;
    /* encode the method return value */

- decodeMethodRetFrom:(NXPortStream *)stream;
    /* decode the return value. */

- (unsigned)sizeOfParams;
    /* Return the size of all parameters including self and sel */
@end

@implementation RemoteMethodInfo

static unsigned hashMethodInfo(const void *info, const void *data) {
    const RemoteMethodInfo      *rm = data;
    return (unsigned)rm->typedesc;      /* depends on the fact its uniqeud */
}
static int isEqualMethodInfo(const void *info, const void *data1, const void *data2) {
    const RemoteMethodInfo      *rm1 = data1;
    const RemoteMethodInfo      *rm2 = data2;
    return (rm1->typedesc == rm2->typedesc);
}

static NXHashTablePrototype proto = {hashMethodInfo, isEqualMethodInfo, NXNoEffectFree, 0}
;

static NXHashTable *allMethodInfos = NULL;
+ initialize {
    if (! knownRemoteMethodInfo) {
        allMethodInfos = NXCreateHashTable(proto, 0, NULL);
        knownRemoteMethodInfo = [RemoteMethodInfo localMethodInfoFor: (Class) [Object clas
s] :@selector(remoteMethodInfo:)];
    }
    return self;
}

+ localMethodInfoFor:(Class)class : (SEL)sel {
    Method      method = class_getInstanceMethod(class, sel);
    RemoteMethodInfo    *previous;
```

```
    if (! method) {
        error("*** localMethodInfoFor:: for '%s' with '%s' return nil\n", [(id) class name
], sel_getName(sel));
        return nil;
    }
    self = [super new];
    typedesc = NXUniqueString(method->method_types);
    previous = NXHashGet(allMethodInfos, self);
    if (previous) {[self free]; return previous; }
    NXHashInsert(allMethodInfos, self);
    return self;
}

- encodeRemotelyFor:(Communication *)communication freeAfterEncoding:(BOOL *)flag {
    return self;
}

- writePortStream: (NXPortStream *) stream {
    [super writePortStream: stream];
    warning("writing a Method:%s\n", typedesc);
    NXWritePortTypes(stream, "%", &typedesc);
    return self;
}

- readPortStream: (NXPortStream *) stream {
    [super readPortStream: stream];
    NXReadPortTypes(stream, "%", &typedesc);
    warning("reading a Method:%s\n", typedesc);
    return self;
}

- encodeMethodParams:(void *)args onto:(NXPortStream *)stream {
    struct objc_method  met;
    unsigned    nb;
    unsigned    index = 2; /* skip result, self and sel */
    int         offset0;
    char        *type;
    met.method_types = (char *)typedesc;
    nb = method_getNumberOfArguments(&met);
    NXWritePortTypes(stream, "i", &nb); /* just for redundancy */
    method_getArgumentInfo(&met, 0, &type, &offset0);
    while (index < nb) {
        char            *type;
        int             offset = 0;
        void            *arg;
        method_getArgumentInfo(&met, index, &type, &offset);
        if (! offset) error("*** encodeMethodParams:onto: cannot extract %d argument for t
ype desc %s\n", index, typedesc);
        arg = ((char *)args)+offset-offset0;
        warning("encodeMethodParams:onto: type=%s value=0x%x\n", type, *(void **)arg);
        NXWritePortTypeInternal(stream, type, arg);
        index++;
    }
    return self;
}

- (void *)decodeMethodParamsFrom:(NXPortStream *)stream {
    struct objc_method  met;
    unsigned    nb;
    unsigned    index = 2;
    unsigned    count;
    void        *args;
    int         offset0;
    char        *type;
```

```
    met.method_types = (char *)typedesc;
    nb = method_getNumberOfArguments(&met);
    NXReadPortTypes(stream, "i", &count);
    if (count != nb) {
        error("decodeMethodParamsFrom: incompatible method params");
        return NULL;
    }
    method_getArgumentInfo(&met, 0, &type, &offset0);
    args = calloc([self sizeOfParams], 1);
    while (index < nb) {
        char            *type;
        int             offset = 0;
        void            *arg;
        method_getArgumentInfo(&met, index, &type, &offset);
        if (! offset) error("*** decodeMethodParamsFrom: cannot extract %d argument for ty
pe desc %s\n", index, typedesc);
        arg = ((char *)args)+offset-offset0;
        NXReadPortTypeInternal(stream, type, arg);
        warning("decodeMethodParamsFrom: type=%s value=0x%x\n", type, *(void **)arg);
        index++;
    }
    return args;
}

- encodeMethodRet:result onto:(NXPortStream *)stream {
//?? -> SN: way to get return types?
    if (typedesc[0] == 'v') return self;
    if (typedesc[0] == 'c') {
        //?? BOGUS, of course
        NXWritePortTypeInternal(stream, "i", &result);
    } else {
        NXWritePortTypeInternal(stream, typedesc, &result);
    }
    return self;
}

- decodeMethodRetFrom:(NXPortStream *)stream {
    id          result = nil;/* important init. for result less than 4 bytes */
//?? -> SN: what iff result more than 4 bytes?
    if (typedesc[0] == 'v') return nil;
    if (typedesc[0] == 'c') {
        //?? BOGUS, of course
        NXReadPortTypeInternal(stream, "i", &result);
    } else {
        NXReadPortTypeInternal(stream, typedesc, &result);
    }
    return result;
}

- (unsigned)sizeOfParams {
    struct objc_method  met;
    met.method_types = (char *)typedesc;
    return method_getSizeOfArguments(&met);
}

@end

@interface Object (Object_RemoteMethodInfo)
- remoteMethodInfo:(SEL)sel;
@end

@implementation Object (Object_RemoteMethodInfo)

- remoteMethodInfo:(SEL)sel {
```

```
    return [RemoteMethodInfo localMethodInfoFor:(Class)[self class] :sel];
}

@end

/*****************      Communication   ***************************/

@implementation Communication

typedef struct _LocalToRemote {
    id                  local;
    RemoteObject        *remote;        /* remote->name == (unsigned)local */
} LocalToRemote;

static void freeLocalToRemote(const void *info, void *data) {
    LocalToRemote       *ltr = data;
    [ltr->remote reallyFree];
    free(data);
}

static NXHashTablePrototype proxyProto;

static id commList = nil;

static receive_enable_proc_t enableProc = NULL;
static port_t replyPort = PORT_NULL;

+ beginListeningOn:(port_t)listenPort enableProc:(receive_enable_proc_t)aProc {
    replyPort = listenPort;
    port_set_backlog(task_self(),listenPort,PORT_BACKLOG_MAX);
    enableProc = aProc;
    if (enableProc) (*enableProc)(replyPort,handleRemoteMessage,YES);
    return self;
}

+ new:(port_t)port timeout:(int)aTimeout {
    NXHashTablePrototype        protol = NXPtrStructKeyPrototype;
    protol.free = freeLocalToRemote;
    if (! commList) commList = [List new];
    self = [super new];
    sendPort = port;
    timeout = aTimeout;
    objectsGivenAway = NXCreateHashTable(protol, 0, NULL);
    allProxies = NXCreateHashTable(proxyProto, 0, NULL);
    [commList addObject:self];
    return self;
}

+ findCommForPort:(port_t)aPort {
    int index = [commList count];
    while (index--)
        if (((Communication *)[commList objectAt:index])->sendPort == aPort)
            return [commList objectAt:index];
    return nil;
}

- free {
    [commList removeObject:self];
    NXFreeHashTable(objectsGivenAway);
    NXFreeHashTable(allProxies);
    return [super free];
}

- beforeEncoding:object onto:(NXPortStream *)stream freeAfterEncoding:(BOOL *)flag {
```

```
    return [object encodeRemotelyFor: stream->communication freeAfterEncoding:flag];
}

- afterDecoding:object from:(NXPortStream *)stream {
    return [object afterPortReading: stream->communication];
}

@end

/*****************      Remote Objects  ************************/

static id localRoot = nil;
static id localRemoteForRoot = nil;

@implementation RemoteObject

+ initialize {
    /* We have to initialize knownRemoteMethodInfo! */
    [RemoteMethodInfo initialize];
    return self;
}

+ newRemote:(unsigned)remoteName withCommunication:(Communication *)communication {
    self = [super new];
    comm = communication;
    name = remoteName;
    if (NXHashInsert(communication->allProxies, self)) error("newRemote: already in table!
");
    return self;
}

+ newLocal:local withCommunication:(Communication *)communication {
    LocalToRemote       pseudo;
    LocalToRemote       *ltr;
    if (! local) return nil;
    if (local == localRoot) return localRemoteForRoot;
    pseudo.local = local;
    ltr = NXHashGet(communication->objectsGivenAway, &pseudo);
    if (ltr) return ltr->remote;
    ltr = malloc(sizeof(LocalToRemote));
    ltr->local = local;
    ltr->remote = [self new];
    ltr->remote->name = (unsigned)local;
    NXHashInsert(communication->objectsGivenAway, ltr);
    return ltr->remote;
}

+ registerLocalRoot:root {
    if (localRoot) error("registerLocalRoot: root registered twice!");
    localRoot = root;
    return (localRemoteForRoot = [self new]);
}

+ messageReceived:(msg_header_t *)msg {
    handleRemoteMessage(msg, NULL);
    return self;
}

- reallyFree {
    [knownSelectors free];
    return [super free];
}

- free {
```

```
    syslog(LOG_ERR, "Remote Object 0x%x received free", self);
    return nil;
}

static unsigned hashProxy(const void *info, const void *data) {
    return ((RemoteObject *)data)->name;
}

static int isEqualProxy(const void *info, const void *data1, const void *data2) {
    return ((RemoteObject *)data1)->name == ((RemoteObject *)data2)->name;
}

static void freeProxy(const void *info, const void *data) {
    [(id)data reallyFree];
}

static NXHashTablePrototype proxyProto = {hashProxy, isEqualProxy, freeProxy, 0};

- (unsigned)remoteObjectName { return name; }

- remoteMethodInfo:(SEL)sel {
    id          res;
    id          args[4];
    if (sel == @selector(remoteMethodInfo:)) return knownRemoteMethodInfo; /* to avoid inf
inite recursion */
    res = [knownSelectors valueForKey:(void *)sel];
    if (res) return res;
//?? -> SN How to fill args cleanly
    bzero(args, sizeof(id)*4);
    {
        Method  method = class_getInstanceMethod((Class)[Object class], @selector(remoteMe
thodInfo:));
        char            *type;
        int             offset2;
        int             offset0;
        SEL             *ref;
        method_getArgumentInfo(method, 0, &type, &offset0);
        method_getArgumentInfo(method, 2, &type, &offset2);
        ref = (SEL *) (((char *)args)+offset2-offset0);
        *ref = sel;
    }
    res = [(id) self forward:@selector(remoteMethodInfo:) :args];
    if (! knownSelectors) knownSelectors = [HashTable newKeyDesc:":"];
    [knownSelectors insertKey:(void *)sel value:res];
    return res;
}

- encodeRemotelyFor:(Communication *)communication freeAfterEncoding:(BOOL *)flag {
    return self;
}

- writePortStream: (NXPortStream *) stream {
    [super writePortStream: stream];
    NXWritePortTypes (stream, "ii", &comm, &name);
    return self;
}

- readPortStream: (NXPortStream *) stream {
    [super readPortStream: stream];
    NXReadPortTypes (stream, "ii", &comm, &name);
    return self;
}

- afterPortReading:(Communication *)communication {
```

```
    if (! comm) {
        id      previous;
        /* it was local for the other guy */
        warning("in afterPortReading - read Remote %d\n", name);
        comm = communication;
        previous = NXHashGet(communication->allProxies, self);
        if (previous) {
            warning("receiving same name=0x%x previous=0x%x self=0x%x\n", name, previous,
self);
            [self reallyFree];
            return previous;
        }
        NXHashInsert(communication->allProxies, self);
        return self;
    } else {
        LocalToRemote   pseudo;
        LocalToRemote   *ltr;
        pseudo.local = (id)name;
        if (! name) {
            if (! localRoot) error("invariant broken in afterPortReading:");
            [self reallyFree];
            return localRoot;
        }
        ltr = NXHashGet(communication->objectsGivenAway, &pseudo);
        if (! ltr) error("afterPortReading:");
        if ((unsigned)ltr->local != name) {
            error("afterPortReading: broken invariant");
        }
        warning("in afterPortReading - converting Remote %d into local 0x%x\n", name, ltr-
>local);
        [self reallyFree];
        return ltr->local;
    }
}

- (unsigned) methodArgSize: (SEL) sel {
    RemoteMethodInfo    *selInfo = [self remoteMethodInfo:sel];
    if (! selInfo) return 0;
    return [selInfo sizeOfParams];
}


- forward:(SEL)sel :(void *)args {
    char                buffer[MSG_SIZE_MAX];
    msg_header_t        *msg = (msg_header_t *)buffer;
    NXAtom              selName = NXUniqueString(sel_getName (sel));
    int                 sequence = generateSequenceNumber();
    NXPortStream        *stream = NXOpenEncodePortStream(msg, sequence, comm);
    RemoteMethodInfo    *selInfo = [self remoteMethodInfo:sel];
    id                  result;
    int                 errorCode;

    if (! selInfo) error("forward:: cannot find remote selector %s", selName);
    warning("entered forward:: self=%d selName=%s\n", name, selName);
    NXWritePortTypes(stream, "@%", &self, &selName);
    [selInfo encodeMethodParams:args onto:stream];
    NXCloseEncodePortStream(stream);
    warning("in forward:: - %d made packet for [0x%x comm:%x %s ...]\n", name, comm, self,
selName);

    remoteAsk(comm->sendPort, msg, comm->timeout);

    /* let's decode the result */
    warning("in forward:: selName=%s received answer\n", selName);
```

```
    stream = NXOpenDecodePortStream(msg, comm);
    NXReadPortTypes (stream, "i", &errorCode);
    if (errorCode) error("forward:: Error occurred during remote execution");
    result = [selInfo decodeMethodRetFrom:stream];
    NXCloseDecodePortStream(stream);
    warning("in forward:: - %d result decoded for (0x%x comm:%x %s ...)\n", name, comm, se
lf, selName);
    return result;
}

@end

/*****************      Object Misc     ***************************/

@interface Object (Object_MakeRemote)
- setAction: (SEL) theSelector;
@end

@implementation Object (Object_MakeRemote_Import)
- encodeRemotelyFor:(Communication *)communication freeAfterEncoding:(BOOL *)flag {
    warning("in encodeRemotelyFor - converting local 0x%x (%s) into remote\n", self, [self
 name]);
    return [RemoteObject newLocal:self withCommunication:communication];
}
- afterPortReading:(Communication *)communication {
    return self;
}

- instantiateObject:(const char *)className {
    return [objc_getClass((char *)className) new];
}

- setOutlet:(const char *)outletName with:dest {
    char        methodString[256];
    SEL         sel;
    strcpy(methodString, "set");
    strcat(methodString, outletName);
    strcat(methodString, ":");
    if (methodString[3] >= 'a' && methodString[3] <= 'z')
        methodString[3] += 'A' - 'a';
    sel = _sel_registerName((char *)NXUniqueString(methodString));
#if 0
    if ([self respondsTo:sel]) {
        [self perform:sel with:dest];
    } else {
        object_setInstanceVariable(self, methodString, dest);
    }
#else
    [self perform:sel with:dest];
#endif
    return self;
}

@end

/*****************      Message transport       ***************************/

#define RO_TELL_MSG_ID (232323)
#define RO_ASK_MSG_ID (323232)
#define RO_REPLY_MSG_ID (233223)

#define DEFAULT_TIMEOUT (15000)

static void remoteReply(port_t rPort, msg_header_t *msg, int timeout);
```

```
static void handleRemoteAsk(msg_header_t *msg, port_t sender, id communication) {
    /*
    ** perform a remote RPC.
    ** This function may call many callouts to enableRemoteListening and
    ** disableRemoteListening  as it recurses.
    ** exceptions may arise.
    */
    NXAtom              selName;
    id                  volatile result = nil;
    NXPortStream        *stream = NXOpenDecodePortStream(msg, communication);
    char                *args;
    id                  self;
    SEL                 sel;
    int                 errorCode = 0;
    RemoteMethodInfo    *selInfo;
    int                 sequence = NXGetPortStreamSequence(stream);
    char                buffer[MSG_SIZE_MAX];

    warning("in remoteAnswer received packet\n");
    NXReadPortTypes(stream, "@%", &self, &selName);
    warning("in remoteAnswer selName=%s\n", selName);
    sel = sel_getUid((char *) selName);
    if (! sel) error("handleRemoteAsk: received message with unknown sel");
    selInfo = [self remoteMethodInfo:sel];
    args = [selInfo decodeMethodParamsFrom:stream];
    NXCloseDecodePortStream(stream);
    if (! args) { errorCode = -2; goto done; }
    NX_DURING
        result = objc_msgSendv(self, sel, [selInfo sizeOfParams], args);
    NX_HANDLER
        ROLog("*** Error while excuting remote message for [0x%x %s ...]\n", self, selName
);
        errorCode = -1;
    NX_ENDHANDLER;
    free(args);

    /* let's encode the result */
  done:
    /* we cannot reuse msg here, regrettably, because if we come from DPSClient, we don't
have an 8K buffer but a copy only big enough for the incoming msg; sigh */
    msg = (msg_header_t *)buffer;
    stream = NXOpenEncodePortStream(msg, sequence, communication);
    NXWritePortTypes (stream, "i", &errorCode);
    [selInfo encodeMethodRet:result onto:stream];
    NXCloseEncodePortStream (stream);
    remoteReply(sender, msg, ((Communication*)communication)->timeout);
    warning("in remoteAnswer made return packet\n");
}

static void handleRemoteTell(msg_header_t *msg, port_t sender, id communication) {
    const char          *selName;
    NXPortStream        *stream = NXOpenDecodePortStream(msg, communication);
    char                *args;
    id                  self;
    SEL                 sel;
    int                 volatile errorCode = 0;
    RemoteMethodInfo    *selInfo;

    warning("in handleRemoteTell received packet\n");
    NXReadPortTypes(stream, "@%", &self, &selName);
    warning("in handleRemoteTell selName=%s\n", selName);
    sel = sel_getUid ((char *) selName);
    if (! sel) error("handleRemoteTell: received message with unknown sel");
```

```
    selInfo = [self remoteMethodInfo:sel];
    args = [selInfo decodeMethodParamsFrom:stream];
    NXCloseDecodePortStream (stream);
    if (! args) { errorCode = -2; goto done; }
    NX_DURING
        objc_msgSendv(self, sel, [selInfo sizeOfParams], args);
    NX_HANDLER
        errorCode = -1;
    NX_ENDHANDLER;
    free(args);
  done:
    if (errorCode) warning("*** Error executing handleRemoteTell %d\n", errorCode);
}

int remoteMessageReceiveCount = 0;
static void handleRemoteMessage(msg_header_t *msg, void *userData) {
    port_t      sender = msg->msg_remote_port;
    id          communication = [Communication findCommForPort:sender];
    if (! communication) {
        syslog(LOG_ERR, "Received message from zombie");
        return;
    }
    remoteMessageReceiveCount++;
    if (msg->msg_id == RO_TELL_MSG_ID) {
        handleRemoteTell(msg,sender, communication);
    } else if (msg->msg_id == RO_ASK_MSG_ID) {
        handleRemoteAsk(msg,sender, communication);
    } else
        syslog(LOG_ERR,"Bogus remote message");
}

#if 0 // Should work, but has never been used/tested
static void remoteTell(port_t target, msg_header_t *msg, port_t sender, int timeout) {
    int err, sndOptions = SEND_SWITCH;

    msg->msg_remote_port = target;
    msg->msg_local_port = sender;
    msg->msg_id = RO_TELL_MSG_ID;
    if (timeout >= 0)
        sndOptions |= SEND_TIMEOUT;
    err = msg_send(msg,sndOptions,timeout);
    if (err) error("remoteTell: cannot send");
}
#endif

static void remoteReply(port_t rPort, msg_header_t *msg, int timeout) {
    int err, sndOptions = SEND_SWITCH;
    msg->msg_remote_port = rPort;
    msg->msg_local_port = PORT_NULL;
    msg->msg_id = RO_REPLY_MSG_ID;
    if (timeout >= 0)
        sndOptions |= SEND_TIMEOUT;
    err = msg_send(msg,sndOptions,timeout);
    if (err) error("remoteReply: cannot send");
}

static nestingLevel=0;


static void remoteAsk(port_t target, msg_header_t *msg, int timeout) {
    int err;
    int volatile sndOptions = SEND_SWITCH;
    int volatile rcvOptions = RCV_NO_SENDERS | RCV_INTERRUPT;
    msg->msg_remote_port = target;
```

```
    msg->msg_local_port = replyPort;
    msg->msg_id = RO_ASK_MSG_ID;
    if (timeout >= 0) {
        sndOptions |= SEND_TIMEOUT;
        rcvOptions |= RCV_TIMEOUT;
    }
    nestingLevel++;
    if (nestingLevel == 1 && enableProc)
        (*enableProc)(replyPort,handleRemoteMessage,NO);
    NX_DURING
        err = msg_send(msg,sndOptions,timeout);
        if (err) error("remoteAsk: cannot send");
        else {
            while (1) {
                msg->msg_size = MSG_SIZE_MAX;
                msg->msg_local_port = replyPort;
                err = msg_receive(msg,rcvOptions,timeout);
                if (err) {
                    ROLog("remoteAsk: cannot receive or timeout\n");
                    NX_RAISE(TIMEOUT_REMOTE_EXCEPTION, "Cannot receive", NULL);
                } else {
                    if (msg->msg_id == RO_REPLY_MSG_ID) {
                        break;
                    } else
                        handleRemoteMessage((msg_header_t *)msg, NULL);
                        //?? IF OUT OF LINE, DEALLOCATE HERE
                }
            }
        }
        nestingLevel--;
        if (!nestingLevel && enableProc)
            (*enableProc)(replyPort,handleRemoteMessage,YES);
    NX_HANDLER
        nestingLevel--;
        if (!nestingLevel && enableProc)
            (*enableProc)(replyPort,handleRemoteMessage,YES);
        NX_RERAISE ();
    NX_ENDHANDLER;
}
```

```
/* This module simply allows NXStrings to be passed across address spaces.
The principle is: to encode a NXString, make a temporary object holding the string, encode
 its characters, free the temporary; to decode a NXString, decode the temporary object, re
place by an immutable string, free the temporary. */

#import "../lowlevel.subproj/NXString.h"

#import "RemoteObject.h"
#import "NXPortStream.h"

#import <string.h>
#import <stdio.h>

@interface _TemporaryStringHolder: Object {
    @public
    NXString    *string;
}
@end

@implementation _TemporaryStringHolder

- writePortStream:(NXPortStream *)stream {
    unsigned    length = [string length];
    NXChar      *chars = malloc(length*sizeof(NXChar));
    [super writePortStream: stream];
    [string setChars:chars];
    NXWritePortTypes(stream, "i", &length);
    NXPortEncodeBytes(stream, (char *)chars, length);
    free(chars);
    return self;
}

- readPortStream: (NXPortStream *) stream {
    unsigned    length;
    NXChar      *chars;
    [super readPortStream: stream];
    NXReadPortTypes(stream, "i", &length);
    chars = malloc(length*sizeof(NXChar));
    NXPortDecodeBytes(stream, (char *)chars, length);
    string = [NXImmutableString newFor:length chars:chars];
    free(chars);
    return self;
}

- afterPortReading:(Communication *)communication {
    NXString    *res = string;
    [self free];
    return res;
}

@end

@interface NXString (NXString_RemoteObject_Coding)
- encodeRemotelyFor:(Communication *)communication freeAfterEncoding:(BOOL *)flag;
@end

@implementation NXString (NXString_RemoteObject_Coding)
- encodeRemotelyFor:(Communication *)communication freeAfterEncoding:(BOOL *)flag {
    _TemporaryStringHolder      *new = [_TemporaryStringHolder new];
```

```
    new->string = self;
    *flag = YES;
    return new;
}

@end
```

```
*/

#import <objc/Object.h>
#import <objc/hashTable.h>
#import <sys/message.h>

extern SEL _sel_registerName(STR key);
    //?? will go away

/*****************       Definitions     ***************************/

typedef struct _NXPortStream {
    msg_header_t        *msg;
//?? DO SIZE TEST LATER
    BOOL        write;          /* writing vs reading */        //REMOVE AFTER DEBUG!
    id          communication;
    char        *chars;
    int         nbchars;
    int         maxchars;
    int         *ints;
    int         nbints;
    int         maxints;
} NXPortStream;

@interface Object (Communication_Calls)
- beforeEncoding: object onto: (NXPortStream *) stream freeAfterEncoding:(BOOL *)flag;
    /* will encode the returned object; if flag is set, returned object will be send 'free
' after encoding */
- afterDecoding: object from: (NXPortStream *) stream;
    /* will replace the decoded object by the returned object */
@end

/******************      global operations       ************************/

extern NXPortStream *NXOpenEncodePortStream(msg_header_t *msg, int sequence, id communicat
ion);

extern NXPortStream *NXOpenDecodePortStream(msg_header_t *msg, id communication);
        /*  buffer is char[MSG_SIZE_MAX];

        If mode is NX_WRITEONLY, creates a NXPortStream, ready for writing, given a physic
al stream on which to actually put the bytes. If mode is NX_READONLY, creates a NXPortStre
am, ready for reading, given a physical stream on which to actually get the bytes.  The ca
ller is responsible for closing physical.  If the file format mismatches right from the st
art with stream format, NULL is returned, otherwise an exception might be raised.

        Iff read, have buffers point to msg */

extern void NXCloseEncodePortStream(NXPortStream *stream);
    /* Copy buffers into message and prepare msg for msg_send;
    free stream */

extern void NXCloseDecodePortStream(NXPortStream *stream);
    /* free stream */

extern int NXGetPortStreamSequence(NXPortStream *stream);

/*****************       Read/Write data         ***************************/

void NXPortEncodeBytes(NXPortStream *stream, const char *bytes, int count);
```

```
*/

#import <objc/Object.h>
#import <objc/hashtable.h>
#import <sys/message.h>

extern SEL _sel_registerName(STR key);
    //?? will go away

/******************      Definitions      ***************************/

typedef struct _NXPortStream {
    msg_header_t        *msg;
//?? DO SIZE TEST LATER
    BOOL        write;          /* writing vs reading */        //REMOVE AFTER DEBUG!
    id          communication;
    char        *chars;
    int         nbchars;
    int         maxchars;
    int         *ints;
    int         nbints;
    int         maxints;
} NXPortStream;

@interface Object (Communication_Calls)
- beforeEncoding: object onto: (NXPortStream *) stream freeAfterEncoding:(BOOL *)flag;
    /* will encode the returned object; if flag is set, returned object will be send 'free
' after encoding */
- afterDecoding: object from: (NXPortStream *) stream;
    /* will replace the decoded object by the returned object */
@end

/******************      global operations      ************************/

extern NXPortStream *NXOpenEncodePortStream(msg_header_t *msg, int sequence, id communicat
ion);

extern NXPortStream *NXOpenDecodePortStream(msg_header_t *msg, id communication);
        /*  buffer is char[MSG_SIZE_MAX];

        If mode is NX_WRITEONLY, creates a NXPortStream, ready for writing, given a physic
al stream on which to actually put the bytes. If mode is NX_READONLY, creates a NXPortStre
am, ready for reading, given a physical stream on which to actually get the bytes.  The ca
ller is responsible for closing physical.  If the file format mismatches right from the st
art with stream format, NULL is returned, otherwise an exception might be raised.

        Iff read, have buffers point to msg */

extern void NXCloseEncodePortStream(NXPortStream *stream);
    /* Copy buffers into message and prepare msg for msg_send;
    free stream */

extern void NXCloseDecodePortStream(NXPortStream *stream);
    /* free stream */

extern int NXGetPortStreamSequence(NXPortStream *stream);

/*****************      Read/Write data      ***************************/

void NXPortEncodeBytes(NXPortStream *stream, const char *bytes, int count);
```

```
void NXPortDecodeBytes(NXPortStream *stream, char *bytes, int count);

extern void NXWritePortTypes(NXPortStream *stream, const char *type, ...);
        /* Restricted to monochar type descriptions;
        Last arguments specify addresses of values to be written.
        It might seem surprising to specify values by address, but this is extremely conve
nient for copy-paste with NXReadPortTypes calls.  A more down-to-the-earth cause for this
passing of addresses is that values of arbitrary size is not well supported in ANSI C for
functions with variable number of arguments. */

extern void NXReadPortTypes(NXPortStream *stream, const char *type, ...);
        /* Restricted to monochar type descriptions;
        Last arguments specify addresses of values to be read.  Expected type is checked a
gainst the type actually present on the stream. */

extern void NXWritePortTypeInternal(NXPortStream *stream, const char *type, const void *da
ta);
extern void NXReadPortTypeInternal(NXPortStream *stream, const char *type, void *data);

/***************        errors          ****************************/

/* several exceptions can occur; the format of all exceptions raised is a label, a message
 string and maybe some extra information */

#define PORTSTREAM_ERROR_BASE 37000     /* less than appkit base */

typedef enum _PortStreamErrors {
    PORTSTREAM_CALLER_ERROR = PORTSTREAM_ERROR_BASE,
    PORTSTREAM_INCONSISTENCY,
    PORTSTREAM_INTERNAL_ERROR
} PortStreamErrors;

@interface Object (Object_PortStream_Calls)
- writePortStream: (NXPortStream *) stream;
- readPortStream: (NXPortStream *) stream;
@end
```

```
/*      NXPortStream.m
        Copyright 1989, NeXT, Inc.
        Bertrand 1989

*/

#import "NXPortStream.h"

#import <stdlib.h>
#import <stdarg.h>
#import <stdio.h>
#import <string.h>
#import <syslog.h>
#import <mach.h>
#import <libc.h>
#import <objc/error.h>
#import <objc/objc-class.h>
#import <objc/objc-runtime.h>

#define DEBUG_LEAKS     0

/*****************       Utilities       ***************************/

static void BUG(const char *str) {
    syslog(LOG_ERR, "*** RemoteObjects: internal error in %s", str);
    NX_RAISE(PORTSTREAM_INTERNAL_ERROR, str, NULL);
}

static void checkExpected(const char *readType, const char *wanted) {
    if (readType == wanted) return;
    if (! readType || strcmp (readType, wanted)) BUG("checkExpected");
}

@implementation Object (Object_PortStream_Calls)
- writePortStream: (NXPortStream *) stream { return self; }
- readPortStream: (NXPortStream *) stream { return self; }
@end

/*****************       Definitions     ***************************/

typedef struct _remote_message_t {
    msg_header_t        header;
    msg_type_t          sequenceType;
    int                 sequence;
    /* following is chars type and chars, int types and ints, etc ... */
} remote_message_t;

static void PortInternalWriteObject(NXPortStream *stream, id object);
    /* write an object without header */
static id PortInternalReadObject(NXPortStream *stream);
    /* read an object without header */

/*****************       Coding data     ***************************/

void NXPortEncodeBytes(NXPortStream *stream, const char *buf, int count) {
    while (stream->nbchars + count > stream->maxchars) {
        stream->maxchars += stream->maxchars + 1;
        stream->chars = realloc(stream->chars, stream->maxchars);
    }
    bcopy(buf, stream->chars + stream->nbchars, count);
    stream->nbchars += count;
}

void NXPortDecodeBytes(NXPortStream *stream, char *buf, int count) {
```

```
    if (count > stream->nbchars) BUG("NXPortDecodeBytes");
    bcopy(stream->chars, buf, count);
    stream->chars += count;
    stream->nbchars -= count;
}

static inline void _NXEncodeChar(NXPortStream *stream, signed char ch) {
    if (stream->nbchars + 1 > stream->maxchars) {
        stream->maxchars += stream->maxchars + 1;
        stream->chars = realloc(stream->chars, stream->maxchars);
    }
    stream->chars[stream->nbchars ++] = ch;
}

static inline signed char _NXDecodeChar(NXPortStream *stream) {
    if (stream->nbchars-- <= 0) BUG("_NXDecodeChar");
    return *(stream->chars ++);
}

static void _NXEncodeInt(NXPortStream *stream, int x) {
    if (stream->nbints + 1 > stream->maxints) {
        stream->maxints += stream->maxints + 1;
        stream->ints = realloc(stream->ints, stream->maxints * sizeof(int));
    }
    stream->ints[stream->nbints++] = x;
}

static int _NXDecodeInt(NXPortStream *stream) {
    if (stream->nbints-- <= 0) BUG("_NXDecodeInt");
    return *(stream->ints++);
}

static void _NXEncodeBool(NXPortStream *stream, BOOL x) {
    _NXEncodeInt(stream, (x) ? 1 : 0);
}

static BOOL _NXDecodeBool(NXPortStream *stream) {
    int         x = _NXDecodeInt(stream);
    if (x && x!=1) BUG("_NXDecodeBool");
    return x;
}

static inline void _NXEncodeFloat(NXPortStream *stream, float x) {
    NXPortEncodeBytes(stream, (char *) &x, sizeof(float));
}

static inline float _NXDecodeFloat(NXPortStream *stream) {
    float       x;
    NXPortDecodeBytes(stream, (char *) &x, sizeof(float));
    return x;
}

static inline void _NXEncodeDouble(NXPortStream *stream, double x) {
    NXPortEncodeBytes(stream, (char *) &x, sizeof(double));
}

static inline double _NXDecodeDouble(NXPortStream *stream) {
    double      x;
    NXPortDecodeBytes(stream, (char *) &x, sizeof(double));
    return x;
}

static void _NXEncodeChars(NXPortStream *stream, const char *str) {
    _NXEncodeBool(stream, (str) ? YES : NO);
```

```
    if (str) {
        int     len = strlen (str);
        _NXEncodeInt(stream, len);
        NXPortEncodeBytes(stream, str, len);
    }
}

static char *_NXDecodeChars(NXPortStream *stream) {
    if (! _NXDecodeBool(stream)) return NULL;
    else {
        int     len = _NXDecodeInt(stream);
        char    *str = (STR) malloc(len+1);
        NXPortDecodeBytes(stream, str, len);
        str[len] = '\0';
        return str;
    }
}

static NXAtom _NXDecodeUniqueString(NXPortStream *stream) {
    char        *new = _NXDecodeChars(stream);
    NXAtom      atom = NXUniqueString(new);
    free(new);
    return atom;
}

/*******        global operations        *******/

NXPortStream *NXOpenEncodePortStream(msg_header_t *msg, int sequence, id communication) {
    NXPortStream        *stream = calloc(sizeof(NXPortStream), 1);
    remote_message_t    *head = (remote_message_t *)msg;
    if (! communication) BUG("NXOpenEncodePortStream");
    stream->communication = communication;
    stream->write = YES;
    stream->msg = msg;

    head->sequenceType.msg_type_name = MSG_TYPE_INTEGER_32;
    head->sequenceType.msg_type_size = sizeof(int) * 8;
    head->sequenceType.msg_type_number = 1;
    head->sequenceType.msg_type_inline = TRUE;
    head->sequenceType.msg_type_longform = FALSE;
    head->sequenceType.msg_type_deallocate = FALSE;
    head->sequence = sequence;

    return stream;
}

NXPortStream *NXOpenDecodePortStream(msg_header_t *msg, id communication) {
    void                *head = ((void *)msg)+sizeof(remote_message_t);
    NXPortStream        *stream = calloc(sizeof(NXPortStream), 1);
    if (! communication) BUG("NXOpenDecodePortStream");
    stream->communication = communication;
    stream->write = NO;
    stream->msg = msg;

    if (msg->msg_simple) {
        msg_type_t              *type;
        type = head;
        if (type->msg_type_name != MSG_TYPE_BYTE)
            BUG("NXOpenDecodePortStream-char");
        stream->nbchars = type->msg_type_number;
        head += sizeof(msg_type_t);
        stream->chars = head;
        head += stream->nbchars;
```

```
        type = head;
        if (type->msg_type_name != MSG_TYPE_INTEGER_32)
            BUG("NXOpenDecodePortStream-int");
        stream->nbints = type->msg_type_number;
        head += sizeof(msg_type_t);
        stream->ints = head;
        head += stream->nbints * sizeof(int);

    } else {
        msg_type_long_t         *type;
        type = head;
        if (type->msg_type_long_name != MSG_TYPE_BYTE)
            BUG("NXOpenDecodePortStream-char");
        stream->nbchars = type->msg_type_long_number;
        head += sizeof(msg_type_long_t);
        stream->chars = ((char **)head)[0];
        head += sizeof(int);

        type = head;
        if (type->msg_type_long_name != MSG_TYPE_INTEGER_32)
            BUG("NXOpenDecodePortStream-int");
        stream->nbints = type->msg_type_long_number;
        head += sizeof(msg_type_long_t);
        stream->ints = ((int **)head)[0];
        head += sizeof(int);

    }
    return stream;
}

void NXCloseEncodePortStream(NXPortStream *stream) {
    void                *head = ((void *)stream->msg)+sizeof(remote_message_t);
    if (! stream->write) BUG("NXCloseEncodePortStream");
    /* We pad chars to a multiple of 4; what a hack! */
    stream->nbchars = ((stream->nbchars + 3)/4)*4;
    stream->chars = realloc(stream->chars, stream->nbchars);

    /* We test for out-of-line */
    stream->msg->msg_simple = (sizeof(remote_message_t)+sizeof(msg_type_t)+stream->nbchars
+sizeof(msg_type_t)+stream->nbints * sizeof(int) < MSG_SIZE_MAX) && (stream->nbchars <= 40
95) && (stream->nbints <= 4095);

    if (stream->msg->msg_simple) {
        msg_type_t              *type;
        type = head;
        type->msg_type_name = MSG_TYPE_BYTE;
        type->msg_type_size = 8;
        type->msg_type_number = stream->nbchars;
        type->msg_type_inline = TRUE;
        type->msg_type_longform = FALSE;
        type->msg_type_deallocate = FALSE;
        head += sizeof(msg_type_t);
        bcopy(stream->chars, head, stream->nbchars);
        head += stream->nbchars;
        free(stream->chars);

        type = head;
        type->msg_type_name = MSG_TYPE_INTEGER_32;
        type->msg_type_size = sizeof(int) * 8;
        type->msg_type_number = stream->nbints;
        type->msg_type_inline = TRUE;
        type->msg_type_longform = FALSE;
        type->msg_type_deallocate = FALSE;
        head += sizeof(msg_type_t);
```

```
            bcopy(stream->ints, head, stream->nbints * sizeof(int));
            head += stream->nbints * sizeof(int);
            free(stream->ints);

        } else {
            msg_type_long_t         *type;
            vm_address_t            buffer;
            type = head;
            type->msg_type_header.msg_type_name = 0;
            type->msg_type_header.msg_type_size = 0;
            type->msg_type_header.msg_type_number = 0;
            type->msg_type_header.msg_type_inline = FALSE;
            type->msg_type_header.msg_type_longform = TRUE;
            type->msg_type_header.msg_type_deallocate = TRUE;
            type->msg_type_long_name = MSG_TYPE_BYTE;
            type->msg_type_long_size = 8;
            type->msg_type_long_number = stream->nbchars;
            head += sizeof(msg_type_long_t);
            if (vm_allocate(task_self(), (vm_address_t *)&buffer, stream->nbchars, 1) != KERN_
 SUCCESS)
                BUG("NXCloseEncodePortStream: can't allocate (chars)");
            bcopy(stream->chars, (char *)buffer, stream->nbchars);
            ((int *)head)[0] = buffer;
            head += sizeof(int);
            free(stream->chars);

            type = head;
            type->msg_type_header.msg_type_name = 0;
            type->msg_type_header.msg_type_size = 0;
            type->msg_type_header.msg_type_number = 0;
            type->msg_type_header.msg_type_inline = FALSE;
            type->msg_type_header.msg_type_longform = TRUE;
            type->msg_type_header.msg_type_deallocate = TRUE;
            type->msg_type_long_name = MSG_TYPE_INTEGER_32;
            type->msg_type_long_size = sizeof(int) * 8;
            type->msg_type_long_number = stream->nbints;
            head += sizeof(msg_type_long_t);
            if (vm_allocate(task_self(), (vm_address_t *)&buffer, stream->nbints * sizeof(int)
 , 1) != KERN_SUCCESS)
                BUG("NXCloseEncodePortStream: can't allocate (ints)");
            bcopy(stream->ints, (char *)buffer, stream->nbints * sizeof(int));
            ((int *)head)[0] = buffer;
            head += sizeof(int);
            free(stream->ints);

        }
        stream->msg->msg_type = MSG_TYPE_NORMAL;
        stream->msg->msg_size = head - (void *)stream->msg;

        free(stream);
    }

    void NXCloseDecodePortStream(NXPortStream *stream) {
        if (stream->write) BUG("NXCloseDecodePortStream");
        if (! stream->msg->msg_simple) {
            void            *head = ((void *)stream->msg)+sizeof(remote_message_t);
            msg_type_long_t *type;
            type = head;
            if (type->msg_type_long_name != MSG_TYPE_BYTE)
                BUG("NXOpenDecodePortStream-char");
            head += sizeof(msg_type_long_t);
            if (vm_deallocate(task_self(), ((int *)head)[0], type->msg_type_long_number) != KE
    RN_SUCCESS)
                BUG("NXCloseDecodePortStream: can't deallocate (chars)");
```

```
            head += sizeof(int);

            type = head;
            if (type->msg_type_long_name != MSG_TYPE_INTEGER_32)
                BUG("NXOpenDecodePortStream-int");
            head += sizeof(msg_type_long_t);
            if (vm_deallocate(task_self(), ((int *)head)[0], type->msg_type_long_number * size
 of(int)) != KERN_SUCCESS)
                BUG("NXCloseDecodePortStream: can't deallocate (ints)");
            head += sizeof(int);

        }
        free(stream);
    }

    int NXGetPortStreamSequence(NXPortStream *stream) {
        remote_message_t    *head = (remote_message_t *)stream->msg;
        return head->sequence;
    }

    /*******         Writing and reading arbitrary data      *******/

    void NXWritePortTypeInternal(NXPortStream *stream, const char *type, const void *data) {
        switch (*type) {
            case 'c': case 'C': {
                char        cc = 0; /* for padding */
                cc = *((char *)data);
                _NXEncodeInt(stream, cc);
                break;
            }
            case 's': case 'S': {
                short       ss = 0; /* for padding */
                ss = *((short *)data);
                _NXEncodeInt(stream, ss);
                break;
            }
            case 'i': case 'I': case 'l': case 'L':
                _NXEncodeInt(stream, *((int *)data));
                break;
            case 'f':
                _NXEncodeFloat(stream, *((float *)data));
                break;
            case 'd':
                _NXEncodeDouble(stream, *((double *)data));
                break;
            case '@':
                PortInternalWriteObject(stream, *((id *)data));
                break;
            case '*':
#if DEBUG_LEAKS
                syslog(LOG_ERR, "*** Remote Objects: un-freeable string encoding: %s", *((char
 **)data));
#endif
                _NXEncodeChars(stream, *((char **)data));
                break;
            case '%':
                _NXEncodeChars(stream, *((NXAtom *)data));
                break;
            case ':': {
                SEL         sel = *((SEL *)data);
                char        *str = NULL;
                if (sel) str = sel_getName (sel);
                _NXEncodeChars(stream, str);
                break;
```

```
            }
            case '!':
                break;
            default: BUG("NXWritePortTypeInternal: unknown type descriptor");
        }
    }

    void NXReadPortTypeInternal(NXPortStream *stream, const char *type, void *data) {
        switch (*type) {
            case 'c': case 'C': {
                signed char         *ptr = data;
                *ptr = _NXDecodeInt(stream);
                break;
            }
            case 's': case 'S': {
                short       *ptr = data;
                *ptr = _NXDecodeInt(stream);
                break;
            }
            case 'i': case 'I': case 'l': case 'L': {
                int         *ptr = data;
                *ptr = _NXDecodeInt(stream);
                break;
            }
            case 'f': {
                float       *ptr = data;
                *ptr = _NXDecodeFloat(stream);
                break;
            }
            case 'd': {
                double      *ptr = data;
                *ptr = _NXDecodeDouble(stream);
                break;
            }
            case '@': {
                id          *ptr = data;
                *ptr = PortInternalReadObject(stream);
                break;
            }
            case '*': {
                char        **ptr = data;
                *ptr = _NXDecodeChars(stream);
                break;
            }
            case '%': {
                NXAtom      *ptr = data;
                *ptr = _NXDecodeUniqueString(stream);
                break;
            }
            case ':': {
                SEL         *ptr = data;
                NXAtom      selName = _NXDecodeUniqueString(stream);
                *ptr = _sel_registerName((char *)selName);
                break;
            }
            case '!':
                break;
            default: BUG("NXReadPortTypeInternal: unknown type descriptor");
        }
    }

    void NXWritePortTypes(NXPortStream *stream, const char *type, ...) {
        va_list     args;
        va_start(args, type);
```

```
        /* We could avoid next line at the cost of more painful debug */
        _NXEncodeChars(stream, type);
        while (*type) {
            NXWritePortTypeInternal(stream, type, va_arg(args, void *));
            type = type++; /* we restrict to monochar type descriptions */
        }
        va_end (args);
    }

    void NXReadPortTypes(NXPortStream *stream, const char *type, ...) {
        NXAtom      readType;
        va_list     args;
        va_start(args, type);
        readType = _NXDecodeUniqueString(stream);
        /* We could avoid next line at the cost of more painful debug */
        checkExpected(readType, type);
        while (*type) {
            NXReadPortTypeInternal(stream, type, va_arg(args, void *));
            type = type++; /* we restrict to monochar type descriptions */
        }
        va_end (args);
    }

    static void PortInternalWriteObject(NXPortStream *stream, id object) {
        BOOL        flag = NO;
        if (! [stream->communication respondsTo: @selector (beforeEncoding:onto:freeAfterEncod
    ing:)]) BUG("PortInternalWriteObject");
        object = [stream->communication beforeEncoding: object onto: stream freeAfterEncoding:
    &flag];
        _NXEncodeBool(stream, (object) ? YES : NO);
        if (object) {
            Class   class = (Class)[object class];
            if (! class) BUG("PortInternalWriteObject: found null class");
            _NXEncodeChars (stream, class->name);
            [object writePortStream: stream];
            _NXEncodeBool(stream, YES);
            if (flag) [object free];
        }
    }

    static id PortInternalReadObject(NXPortStream *stream) {
        id          object;
        if (! _NXDecodeBool(stream)) return nil;
        else {
            NXAtom  className = _NXDecodeUniqueString (stream);
            Class   class = (Class) objc_getClass ((char *) className);
            if (! class) BUG("PortInternalReadObject: class not loaded");
            /* explicit class initialization */
            (void) [(id) class self];
            object = class_createInstance (class, 0);
            [object readPortStream: stream];
            [object awake];
            if (! [stream->communication respondsTo: @selector (afterDecoding:from:)]) BUG("Po
    rtInternalWriteObject-1");
            object = [stream->communication afterDecoding: object from: stream];
            if (! _NXDecodeBool(stream)) BUG("PortInternalReadObject-2");
            return object;
        }
    }
```

We claim:

1. A method for sending an object oriented programming language based message having dynamic binding from a first object in a first process to a second object in a second process, said method comprising the steps of:

transmitting, using a first processing means, said object oriented programming language based message to a first proxy in said first process;

using said first proxy and said first processing means, encoding said object oriented programming language based message into an operating system based message at run time;

transmitting said operating system based message to said second process in said second processing means at run time;

decoding, using a second process, said operating system based message into a language based message;

transmitting, using said second processing means, said object oriented programming language based message to said second object in said second process;

executing said object oriented programming language based message by said second object in said second process.

2. The method of claim 1 further including the steps of:

said second object in said second process and generating an object oriented programming language based result;

encoding, using said second processing means, said object oriented programming language based result into an operating system based result at run time;

transmitting, using said second processing means, said operating system based result to said first process at run time;

decoding said operating system based result into an object oriented programming language based result at run time, using said first processing means;

transmitting, using said first processing means, said object oriented programming language based result to said first object.

3. The method of claim 1 wherein said object oriented programming language based message comprises a method and an argument.

4. The method of claim 1 wherein said second object executes said method on said argument when executing said message.

5. The method of claim 1 wherein the step of executing said object oriented programming language based message further includes the steps of:

said second object determining, using said second processing means, whether additional information is needed to execute said object oriented programming language based message;

said second object generating, using said second processing means, an object oriented programming language based query if it is determined that additional information is needed;

encoding, using said second processing means, said object oriented programming language based query into an operating system based query at run time if it is determined that additional information is needed;

transmitting said operating system based query to said first process at run time, using said second processing means if it is determined that additional information is needed;

decoding, using said first processing means, said operating system based query into an object oriented programming language based query at run time if it is determined that additional information is needed;

transmitting, using said first processing means, said object oriented programming language based query to said first object if it is determined that additional information is needed.

6. The method of claim 5 further including the steps of:

said first object generating, using said first processing means, an object oriented programming language based reply to said object oriented programming language based query;

encoding said object oriented programming language based reply into an operating system based reply at run time, using said first processing means;

transmitting, using said first processing means, said operating system based reply to said second process at run time;

decoding, using said second processing means, said operating system based reply into an object oriented programming language based reply at run time;

transmitting, using said second processing means, said object oriented programming language based reply to said second object.

7. The method of claim 6 wherein said first processing means and said second processing means are the same processing means.

8. The method of claim 1 wherein said object oriented programming language based message comprises an objective C message.

9. The method of claim 1 wherein said operating system based message comprises a Mach message.

10. The method of claim 1 wherein said first proxy represents said second object.

11. A method for sending an object oriented programming language based message having dynamic binding from a first object in a first process to a second object in a second process, said method comprising the steps of:

transmitting, using a first processing means, said object oriented programming language based message to a first proxy in said first process;

using said first proxy and said first processing means, encoding said object oriented programming language based message into an operating system based message at run time;

transmitting, using said first processing means, said operating system based message to said second process at run time;

decoding, using said second processing means, said operating system based message into an object oriented programming language based message at run time;

transmitting, using said second processing means, said object oriented programming language based message to said second object;

said second object generating an object oriented programming language based query, using said second processing means;

creating, using said second processing means, a second proxy in said second process;

transmitting, using said second processing means, said object oriented programming language based query to said second proxy;

using said second proxy and said second processing means, encoding said object oriented programming language based query into an operating system based query at run time;

transmitting, using said second processing means, said operating system based query to said first process at run time;

decoding, using said first processing means, said operating system based query into an object oriented programming language based query at run time;

transmitting, using said first processing means, said object oriented programming language based query to said first object;

said first object generating an object oriented programming language based reply, using said first processing means;

encoding, using said first processing means, said object oriented programming language based reply into an operating system based reply at run time;

transmitting, using said first processing means, said operating system based reply to said second process at run time;

decoding, using a second processing means, said operating system based reply into an object oriented programming language based reply at run time;

transmitting, using said second processing means, said object oriented programming language based reply, using said second processing means, and generating an object oriented programming language based result;

encoding, using said second processing means, said object oriented programming language based result into an operating system based result at run time;

transmitting, using said second processing means, said operating system based result to said first process at run time;

decoding, using said first processing means, said operating system based result into an object oriented programming language based result;

transmitting, using said first processing means, said object oriented programming language based result to said first object.

**12.** The method of claim **11** wherein said object oriented programming language based message comprises a method and an argument.

**13.** The method of claim **12** wherein said second object executes said method on said argument when executing said message.

**14.** The method of claim **11** wherein said first process and said second process are located on first and second computers respectively.

**15.** The method of claim **11** wherein said object oriented programming language based message comprises an objective C message.

**16.** The method of claim **11** wherein said operating system based message comprises a Mach message.

**17.** The method of claim **11** wherein said first proxy represents said second object.

**18.** The method of claim **11** wherein said second proxy represents said first object.

**19.** A method for sending, in a C environment with minimal run time support, an object oriented programming language based message having dynamic binding from a first object in a first process to a second object in a second process, said method comprising the steps of:

transmitting, using a first processing means implementing said C environment, said object oriented programming language based message to a first proxy in said first process;

using said first proxy and said first processing means, encoding said object oriented programming language based message into an operating system based message at run time;

transmitting said operating system based message to said second process at run time;

decoding, using a second processing means implementing said C environment, said operating system based message into a language based message;

transmitting, using said second processing means, said object oriented programming language based message to said second object.

**20.** The method of claim **19** further including the steps of:

said second object executing said object oriented programming language based message, using said second processing means, and generating an object oriented programming language based result;

encoding, using said second processing means, said object oriented programming language based result into an operating system based result at run time;

transmitting, using said second processing means, said operating system based result to said first process at run time;

decoding said operating system based result into an object oriented programming language based result at run time, using said first processing means;

transmitting, using said first processing means, said object oriented programming language based result to said first object.

**21.** The method of claim **20** wherein the step of executing said object oriented programming language based message further includes the steps of:

said second object determining, using said second processing means, whether additional information is needed to execute said object oriented programming language based message;

said second object generating, using said second processing means, an object oriented programming language based query if it is determined that additional information is needed;

encoding, using said second processing means, said object oriented programming language based query into an operating system based query at run time if it is determined that additional information is needed;

transmitting said operating system based query to said first process at run time, using said second processing means if it is determined that additional information is needed;

decoding, using said first processing means, said operating system based query into an object oriented programming language based query at run time if it is determined that additional information is needed;

transmitting, using said first processing means, said object oriented programming language based query to said first object if it is determined that additional information is needed.

**22.** The method of claim **21** further including the steps of:

said first object generating, using said first processing means, an object oriented programming language based reply to said object oriented programming language based query;

encoding said object oriented programming language based reply into an operating system based reply at run time, using said first processing means;

transmitting, using said first processing means, said operating system based reply to said second process at run time;

decoding, using said second processing means, said operating system based reply into an object oriented programming language based reply at run time;

transmitting, using said processing means, said object oriented programming language based reply to said second object.

23. The method of claim 21 wherein said operating system based message comprises a Mach message.



24. The method of claim 21 wherein said first processing means and said second processing means are the same processing means.

* * * * *

# EXHIBIT D

U 7172161

# THE UNITED STATES OF AMERICA

## TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office

February 25, 2009

THIS IS TO CERTIFY THAT ANNEXED HERETO IS A TRUE COPY FROM THE RECORDS OF THIS OFFICE OF:

U.S. PATENT: *5,566,337*
ISSUE DATE: *October 15, 1996*

By Authority of the
Under Secretary of Commerce for Intellectual Property
and Director of the United States Patent and Trademark Office

P. SWAIN
Certifying Officer

US005566337A

# United States Patent [19]

Szymanski et al.

[11] Patent Number: 5,566,337

[45] Date of Patent: Oct. 15, 1996

[54] METHOD AND APPARATUS FOR DISTRIBUTING EVENTS IN AN OPERATING SYSTEM

[75] Inventors: Steven J. Szymanski, Cupertino; Thomas E. Saulpaugh, San Jose; William J. Keenan, Redwood City, all of Calif.

[73] Assignee: Apple Computer, Inc., Cupertino, Calif.

[21] Appl. No.: 242,204

[22] Filed: May 13, 1994

[51] Int. Cl.[6] ..... G06F 9/00

[52] U.S. Cl. ..... 395/733; 395/650; 395/700

[58] Field of Search ..... 395/650, 725, 395/700, 775

[56] References Cited

U.S. PATENT DOCUMENTS

| | | | |
|---|---|---|---|
| 5,155,842 | 10/1992 | Rubin | 395/182.2 |
| 5,237,684 | 8/1993 | Record et al. | 395/650 |
| 5,291,608 | 3/1994 | Flurry | 395/725 |
| 5,305,454 | 4/1994 | Record et al. | 395/650 |
| 5,321,837 | 6/1994 | Daniel et al. | 395/650 |
| 5,355,484 | 10/1994 | Record et al. | 395/650 |
| 5,430,875 | 7/1995 | Ma | 395/650 |

FOREIGN PATENT DOCUMENTS

| | | |
|---|---|---|
| 0528222 | 2/1993 | European Pat. Off. . |
| WO91/03017 | 3/1991 | WIPO . |

OTHER PUBLICATIONS

IBM: 'OS/2 2.0 Presentation Manager Programming Guide', Mar. 1992, QUE, USA, p. 31–5, last paragrph, p. 31–6, paragraph 3.

*Primary Examiner*—Jack B. Harvey
*Assistant Examiner*—Sumati Lefkowitz
*Attorney, Agent, or Firm*—Burns, Doane, Swecker & Mathis

[57] ABSTRACT

In a computer including an operating system, an event producer for generating an event and detecting that an event has occurred in the computer and an event consumer which need to be informed when events occur in the computer, a system for distributing events including a store for storing a specific set of events of which the at least one event consumer is to be informed, an event manager control unit for receiving the event from the event producer, comparing the received event to the stored set of events, and distributing an appropriate event to an appropriate event consumer, and a distributor for receiving the event from the control unit and directing the control unit to distribute an appropriate event to an appropriate event consumer.

**24 Claims, 10 Drawing Sheets**

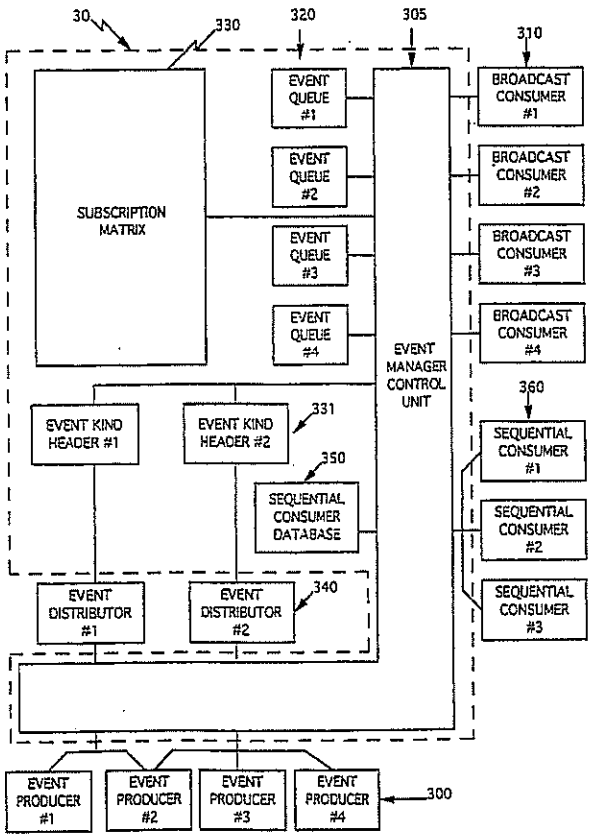
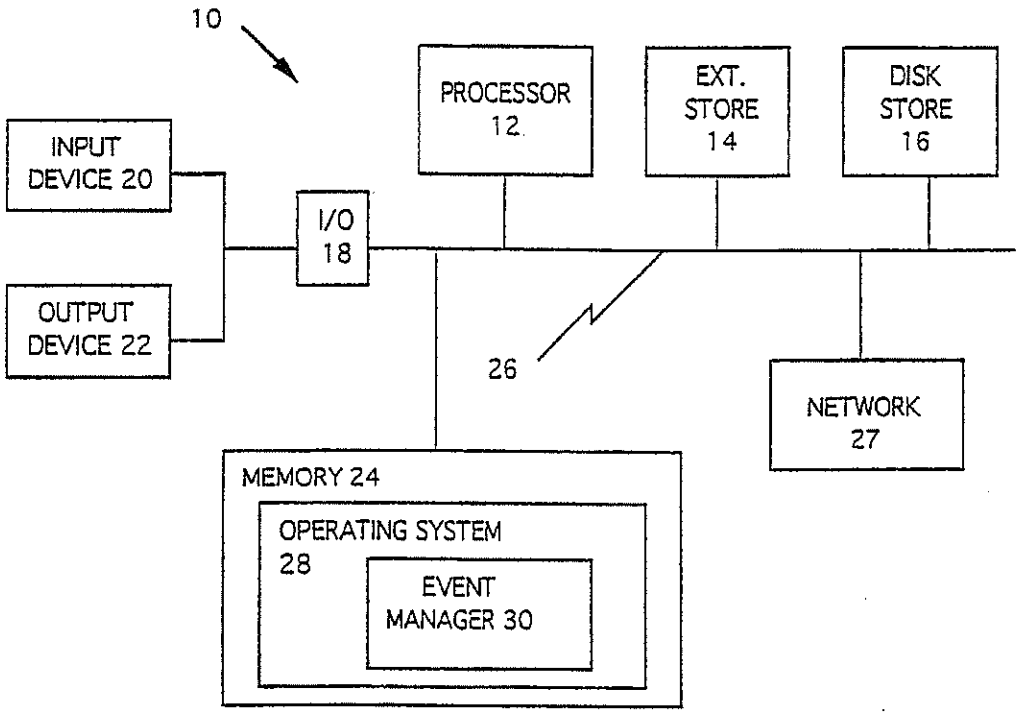


FIG. 1



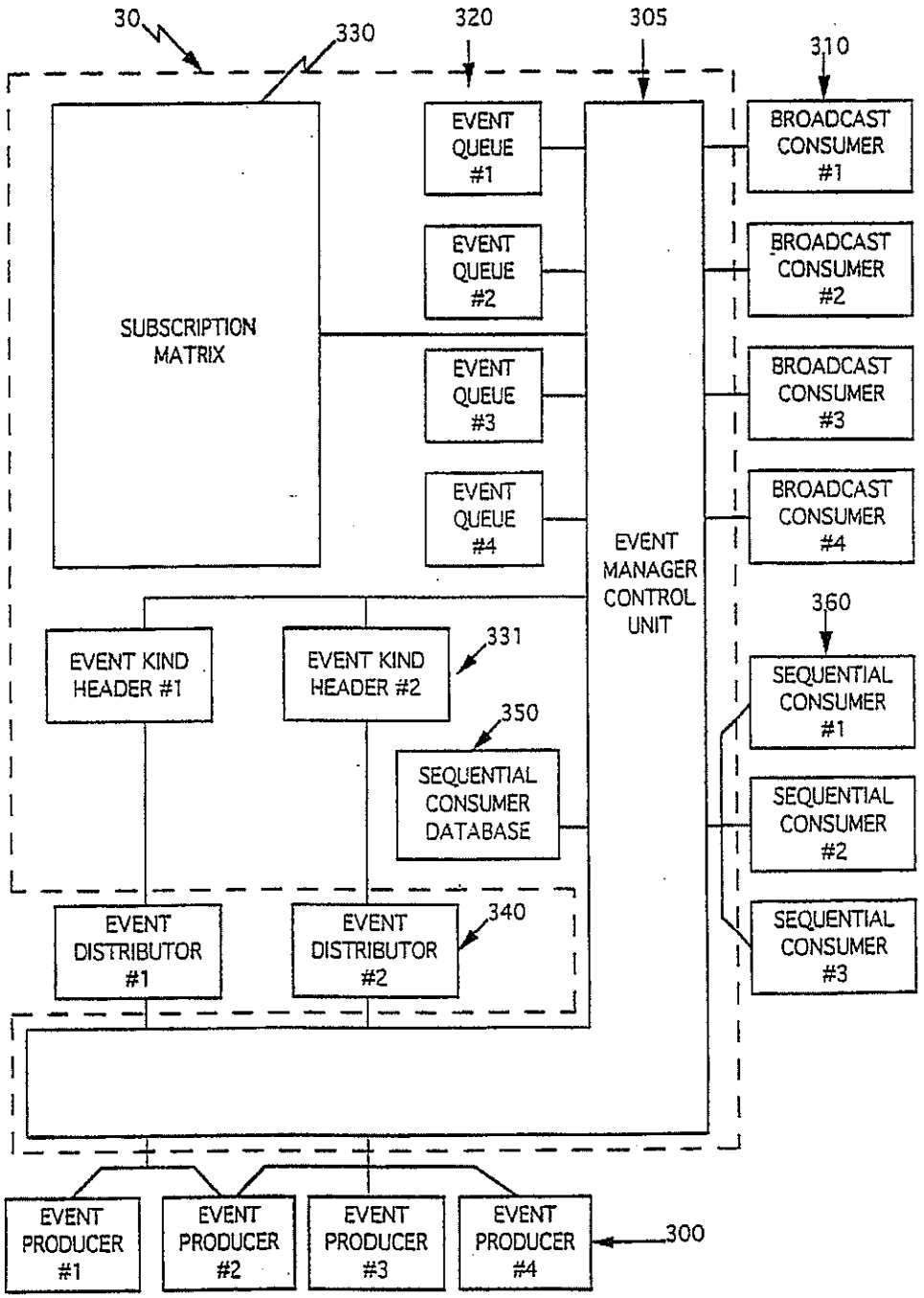



FIG. 2

FIG. 3

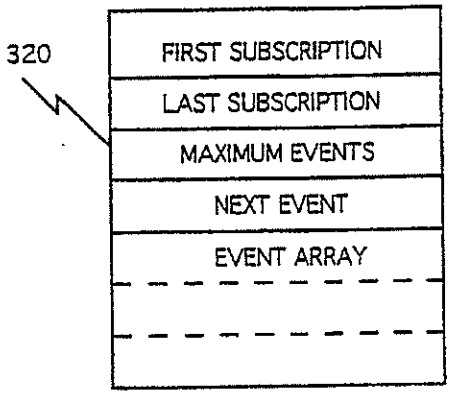



FIG. 6

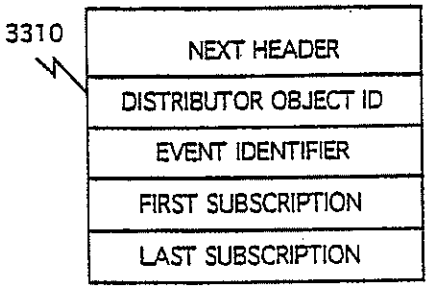



FIG. 5a

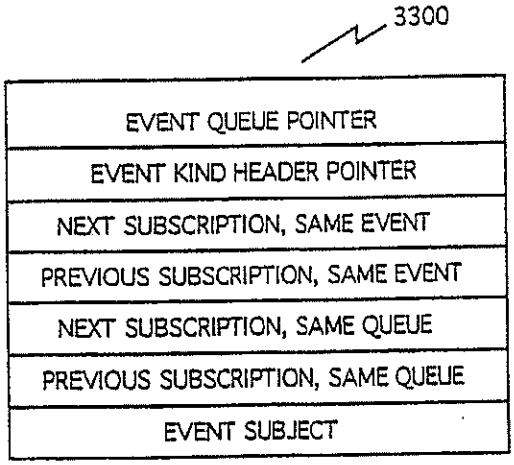



FIG. 7

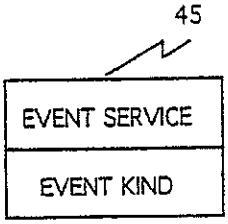



FIG. 8

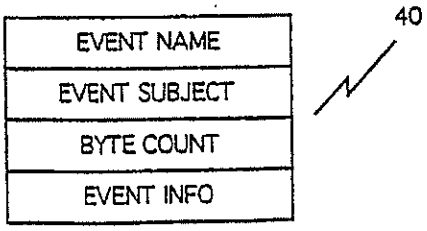
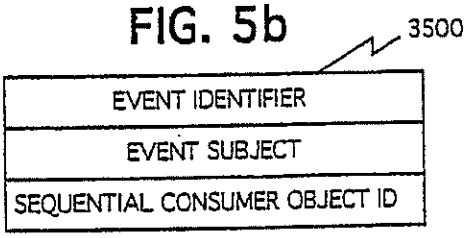



FIG. 5b



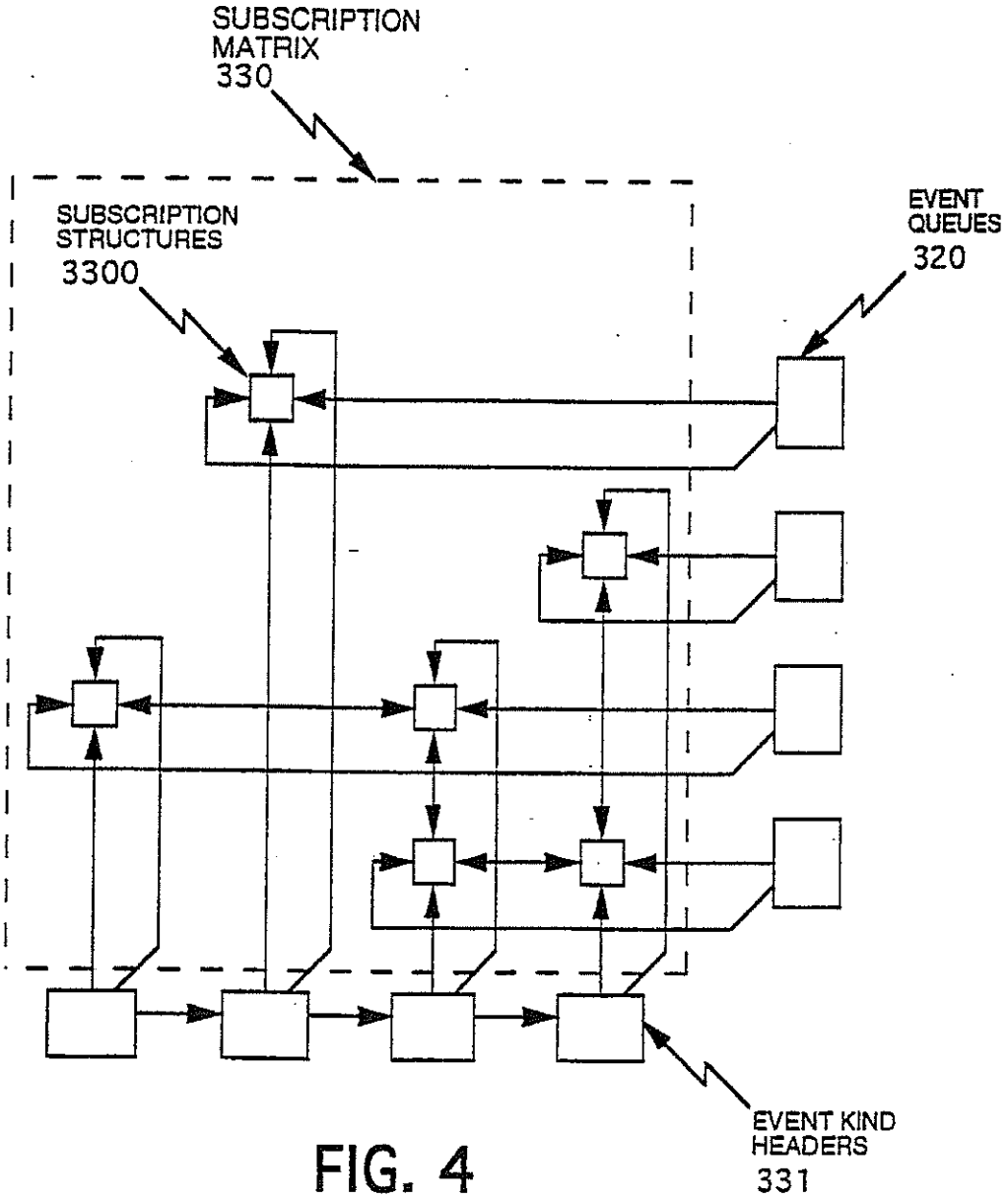



FIG. 4

FIG. 9A

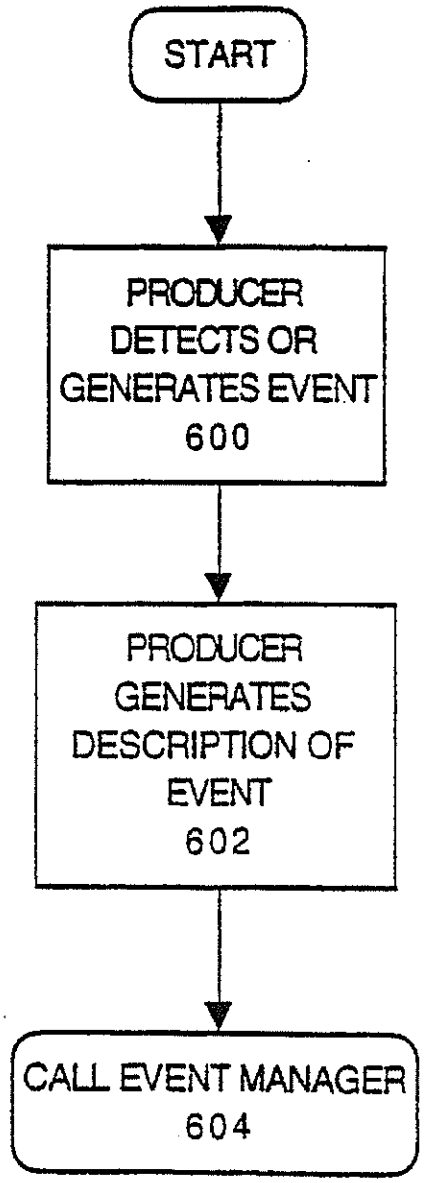


FIG. 9B

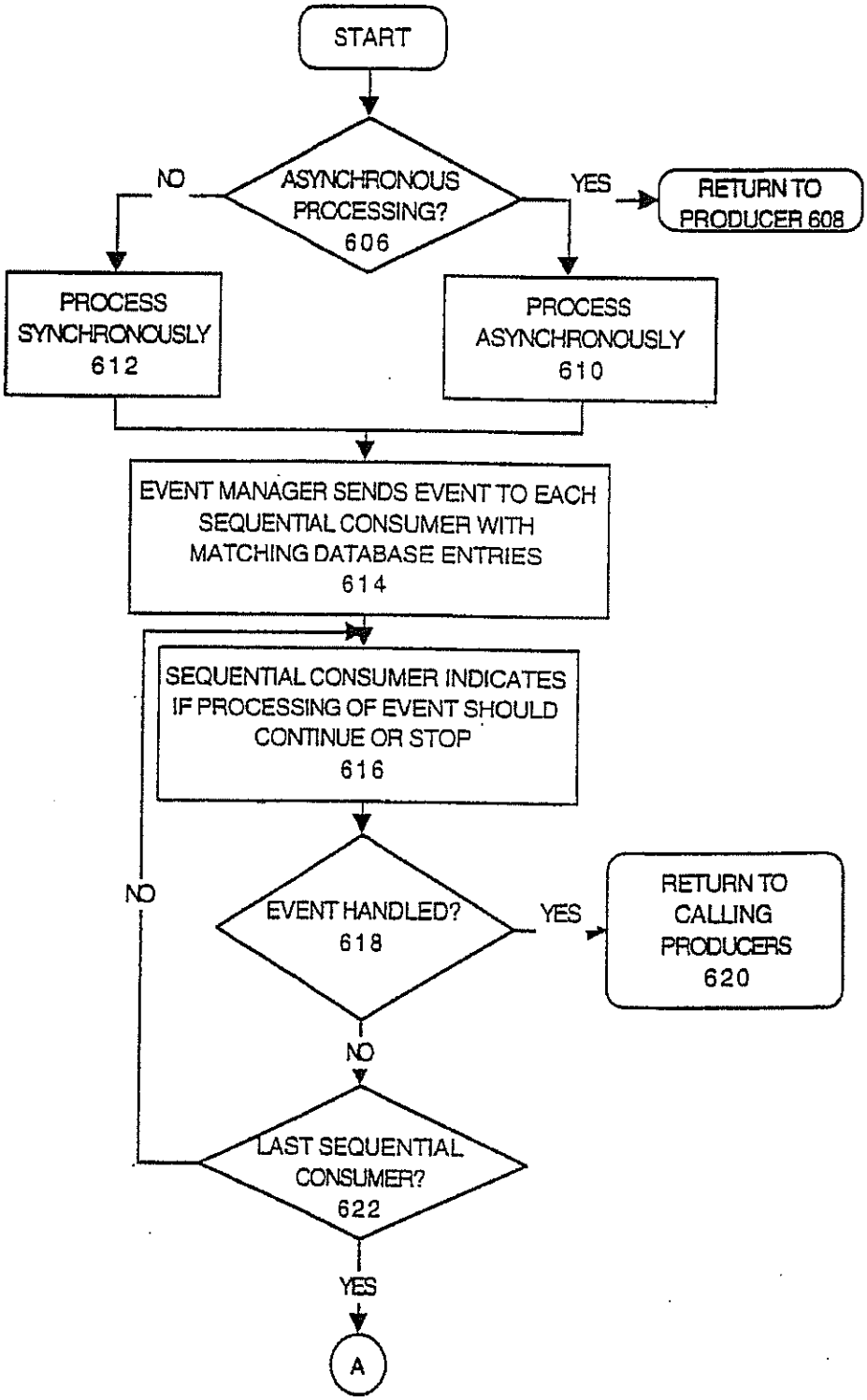


FIG. 9C

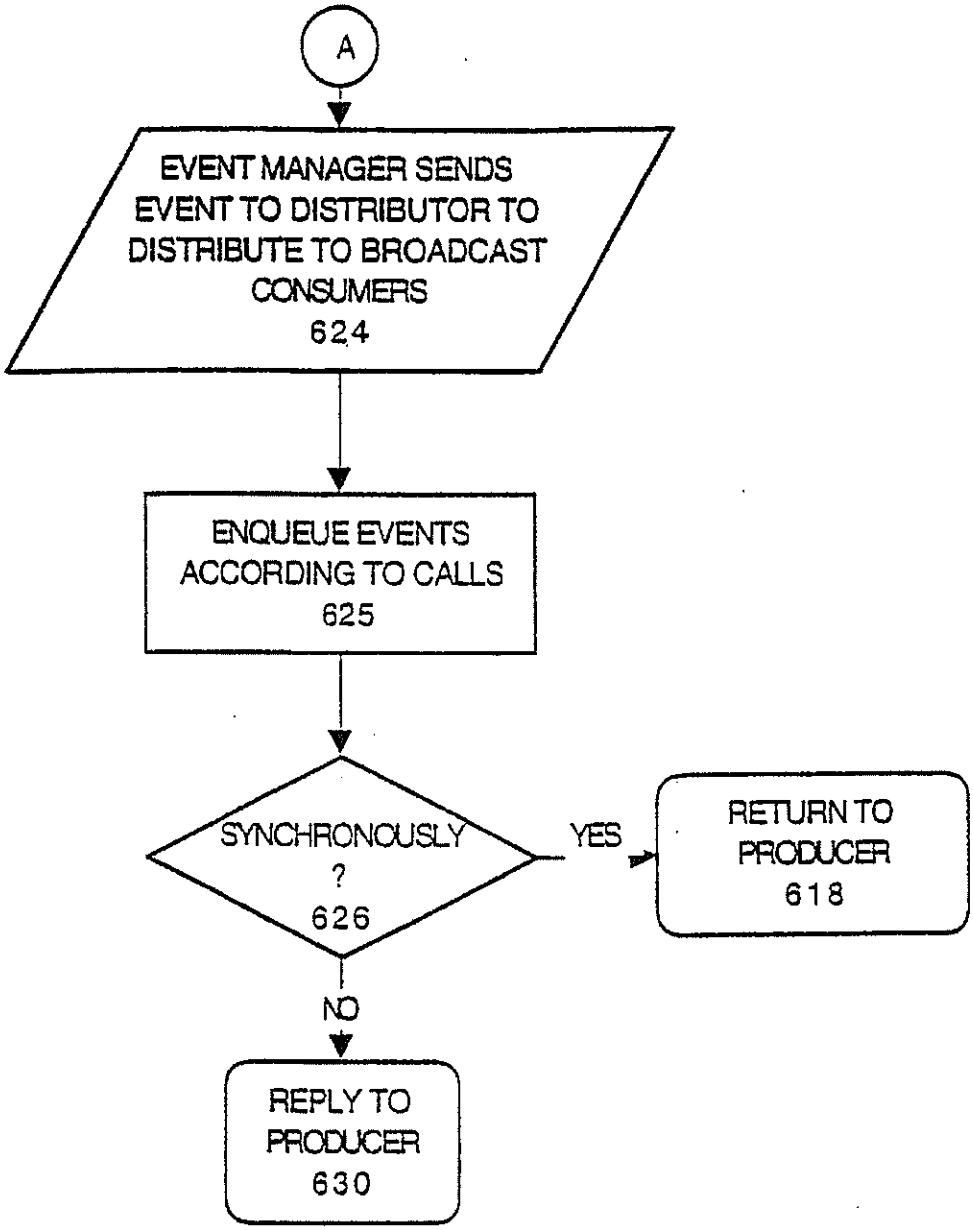


FIG. 9D

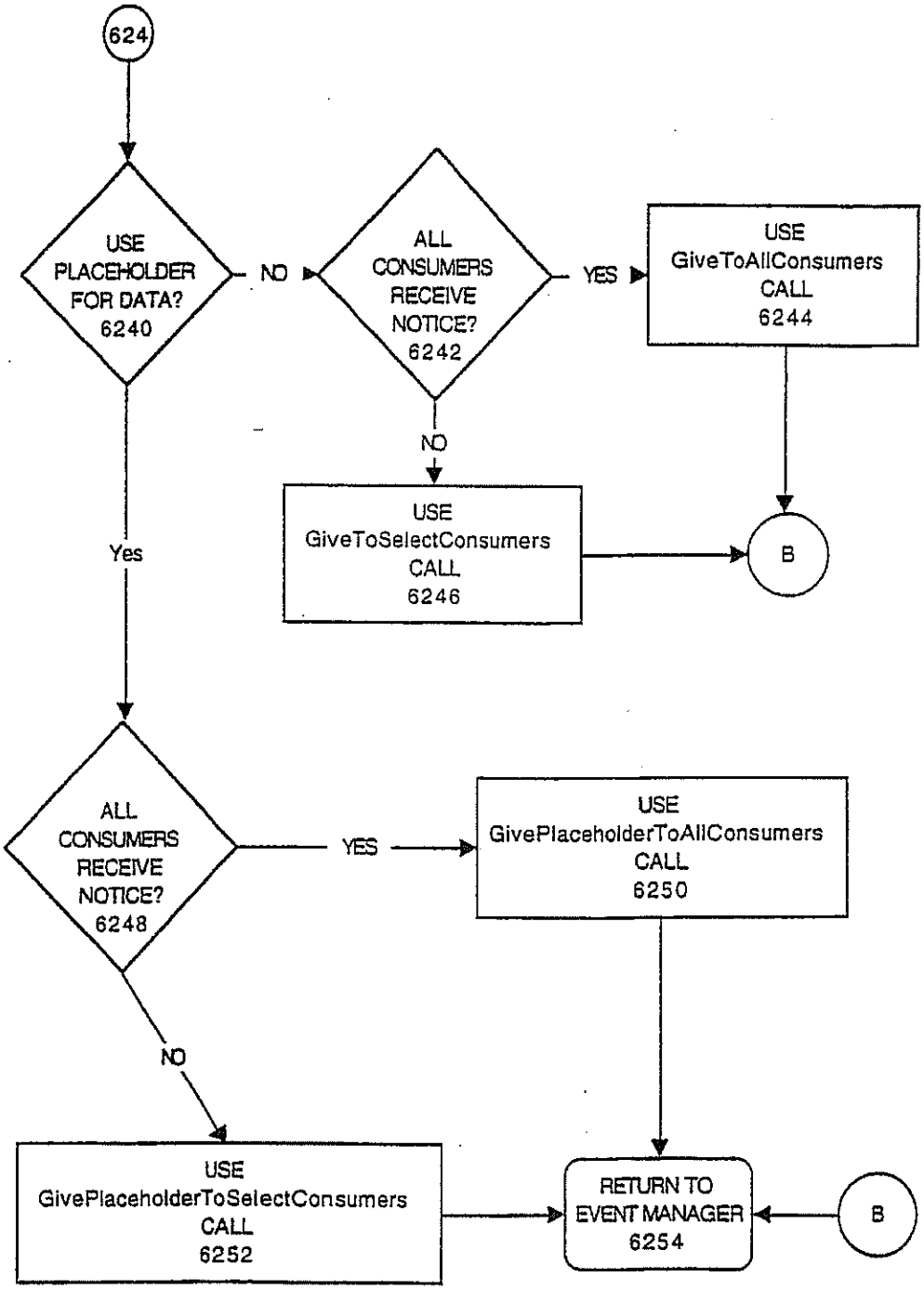


FIG. 10

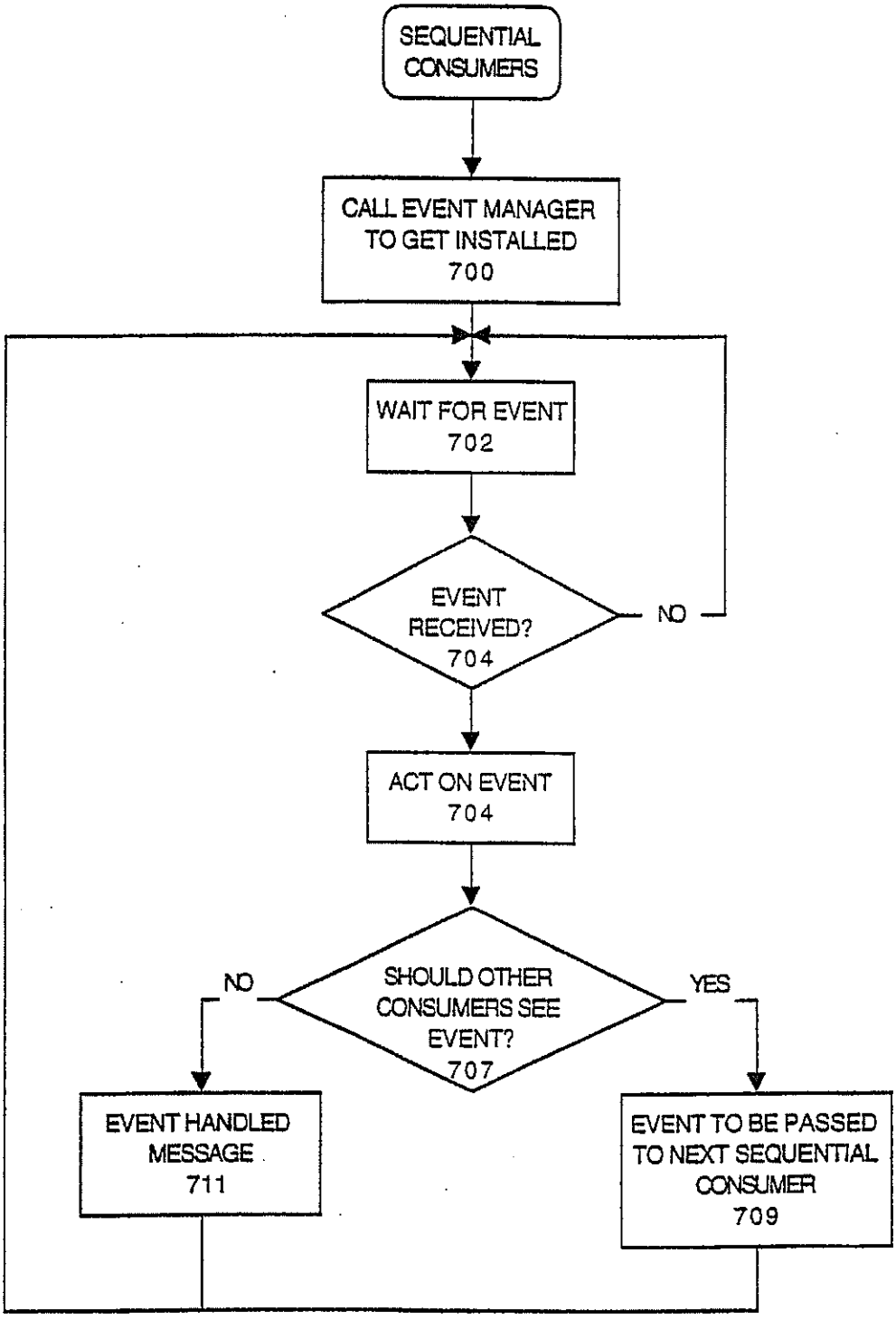
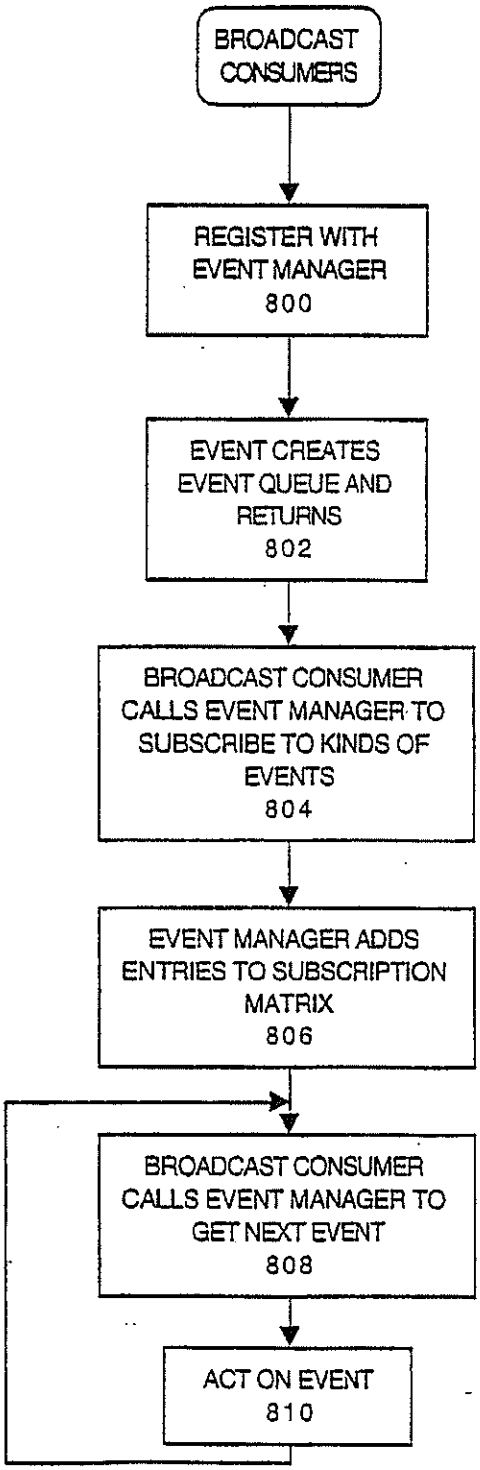


FIG. 11



# METHOD AND APPARATUS FOR DISTRIBUTING EVENTS IN AN OPERATING SYSTEM

## CROSS-REFERENCE TO RELATED APPLICATIONS

The present application is related to a patent application No. 08/245,141 entitled "Method and Apparatus for Handling Requests Regarding Information Stored in A File System", in the name of Steven James Szymanski and Bill Monroe Bruffey, filed on May 13, 1994, herein incorporated by reference.

## BACKGROUND

The present invention is directed to a method and apparatus for distributing information about events occurring in a computer, and in particular an event manager which manages the distribution of those events to the appropriate entities within the computer.

For purposes of this description, an event is any occurrence in a computer of which software programs running on that computer or on a connected computer might need to be informed. Events may include occurrences such as, for example, a keystroke, a mouse click, disk insertion and ejection, network connection and disconnection, the computer entering a "sleep mode" shutdown, a window uncovered (i.e., the contents of the window need to be redisplayed), a new file created, a directory renamed, the contents of file changed, and the tree space on a volume changed, etc.

Interrupts and error conditions may also be counted as atypical examples of events. In particular, interrupts need to be handled by a program so an event manager is an inappropriate solution. However, the code which does handle the interrupt might generate an event based on the interpretation of the interrupt. For example, the computer might generate an interrupt when the user inserts a floppy disk. The interrupt itself is unlikely to be propagated by the event manager, but it would be reasonable for the interrupt handler to produce a "disk inserted" event. Error conditions are similar. Most of the time it is necessary for one of the computer programs on the system to handle the error, therefore more direct point to point mechanisms are appropriate. However, there are kinds of errors which are more advisory in nature which would be appropriate to be sent via events. For example, some portable computers take various actions to reduce power consumption when the battery gets low. It would be appropriate to produce an event called "battery low" to inform all software programs of the condition, and have all of the software which can reduce power consumption consume these events.

Currently, known operating systems all have some type of mechanism for managing the events that occur within the computer. However, these mechanisms use a point-to-point method of managing the events. That is, the entities producing or detecting events distribute the events to the entities using the events. To accomplish this, all of the entities producing or detecting events must know which entities they must notify when a particular event is generated within the computer. This configuration is very cumbersome and inefficient. Further, it is resource intensive since all entities producing or detecting events must have information on all the events they produce or detect and also on all the entities interested in those events. This information is both extensive and constantly changing, causing modifications to be difficult.



Further, point-to-point mechanisms lack flexibility. Under point-to-point schemes, if there is a new consumer of an event, a new version of the producer must be released which knows about the new consumer. Or if a new kind of event becomes necessary, a new version of the event manager must be released which knows how to distribute the new kind of event.

It is desirable to provide an apparatus for efficiently dealing with all kinds of events in an operating system and for distributing information regarding specific kinds of events to programs which require such information. To this end, it is also desirable to improve the system performance and reduce the resources required to distribute such information. To meet these goals, it is desirable to provide an apparatus for managing events in which communication between the event producers and consumers is facilitated without requiring each event producer to be aware of all of the event consumers.

## BRIEF STATEMENT OF THE INVENTION

In accordance with the present invention, the foregoing objectives, as well as others, are achieved through centralization of event management, and in particular, by providing an event manager for handling the distribution of events within the computer.

According to one embodiment, in a computer including at least one event producer for detecting that an event has occurred in the computer and generating an event and at least one event consumer which need to be informed when events occur in the computer, a system is provided for distributing information about events. The system includes storing means for storing a specific set of events of which the event consumers are to be informed, an event manager control means for receiving the event from the event producer, comparing the received event to the stored set of events, and distributing an appropriate event to an appropriate event consumer, and a distributor for receiving the event from the control means and directing the control means to distribute an appropriate event to an appropriate event consumer.

According to another embodiment, a system is provided for distributing events occurring in a computer. The system comprises event producers for detecting that an event has occurred in the computer, generating an event, and generating a description of the event and event consumers which need to be informed when events occur in the computer, the event consumers comprising a first and a second class of consumers. The system further comprises storing means for storing a specific set of events of which the event consumers am to be informed, event manager control means for receiving the event from the event producers and comparing the received event to the stored set of events, distributor means, responsive to the event control means, for deciding if an event should be passed to an event consumer. The event manager control means comprises first means for sending an event to appropriate event consumers of a first type in accordance with the stored set of events, and second means for sending the event to appropriate event consumers of a second type responsive to the distributor means.

According to another embodiment, a method is provided for distributing events occurring in a computer. The method comprises the steps of determining that an event has been detected by an event producer in the computer, storing, in a storing means, a specific set of events of which an event consumer is to be informed, receiving the event in an event

control means from the event producer and comparing the received event to the stored set of events. The method further comprises receiving the event in a distributor means from the control means, directing the control means to distribute an appropriate event to an appropriate event consumer, and distributing, via the control means, an appropriate event to an appropriate event consumer.

Still other objects, features and attendant advantages of the present invention will become apparent to those skilled in the art from a reading of the following detailed description of the embodiments constructed in accordance therewith, taken in conjunction with the accompanying drawings.

## BRIEF DESCRIPTION OF THE DRAWINGS

The present invention will now be described in more detail with reference to preferred embodiments of the method and apparatus, given only by way of example, and with reference to the accompanying drawings, in which:

FIG. **1** is a block diagram of an exemplary computer on which the present invention can be implemented;

FIG. **2** is a block diagram of the architecture for the event manager according to one embodiment of the present invention;

FIG. **3** is an exemplary embodiment of the event queue data structure according to the present invention;

FIG. **4** is an exemplary embodiment of the subscription matrix according to the present invention;

FIG. **5***a* is an exemplary embodiment of the subscription data structure according to the present invention;

FIG. **5***b* is an exemplary embodiment of the sequential consumer entry structure according to the present invention;

FIG. **6** is an exemplary embodiment of the event kind header data structure according to the present invention;

FIG. **7** is an exemplary embodiment of the event name data structure according to the present invention;

FIG. **8** is an exemplary embodiment of the event data structure according to the present invention;

FIGS. **9**A, **9**B, **9**C and **9**D are flowchart illustrating the event handling process from the event producers' and the event distributors' point of view according to one embodiment of the present invention;

FIG. **10** is a flowchart illustrating the event handling process from the sequential consumers' point of view according to one embodiment of the present invention; and

FIG. **11** is a flowchart illustrating the event handling process from the broadcast consumers' point of view according to one embodiment of the present invention.

## DETAILED DESCRIPTION

In general, the invention recognizes the need for efficient communications between different entities within the computer concerning events occurring within the computer. In particular, communications are required to inform entities within the computer about the events produced by other entities. The method required to handle these communications is complicated by the fact that the entities involved do not know the identity of the other entities. The method is further complicated by the fact that the identity of the entities needing to know about events and the lists of events which can occur are subject to constant change.

One goal for the event manager according to the present invention is to provide a common service which supports a majority of these kinds of communications. The information which needs to be communicated is referred to as the event. According to one embodiment, the event is described by three pans, an event identifier which indicates the kind of event, an event subject, which identifies the entity which the event happened to, and event information which describes how the event occurred. The entities within the computer which are the sources of the information are referred to as the producers of the event. In particular, an event producer is any software on a computer that is responsible for generating an event or for detecting that the computer hardware has generated an event. The event producer then generates a description for each event it produces or detects. The entities within the computer which need to receive the information are referred to as consumers of the event. In particular, an event consumer is any program that needs to be informed when an event has occurred and needs to be informed of the description of the event. Any intermediate service which moderates the connection between the producers and consumers of an event is referred to as the distributor of the event.

According to one embodiment of the present invention, there are two classes of event consumers which differ in their relationships to other consumers of an event, namely broadcast consumers and sequential consumers. Broadcast consumers have no relationship with other consumers. They do not need to know if other consumers exist, nor in what order consumers are informed of the event, as long as they themselves are eventually informed. Sequential consumers, on the other hand, have very definite relationships with other consumers. They require that no other consumer be told about an event while they themselves are still processing it, and they require the ability to influence when in the sequence they receive the event. In addition, many sequential consumers require the ability to modify the event itself, and even block an event from being received by other consumers. Consumers are defined as sequential or broadcast based on whether they have to react to an event without fail. In particular, a sequential consumer must react to the kinds of events in which it is interested and so a distributor should not withhold those events from the sequential consumer. For example, a communications program would be a sequential consumer of events which notify of the dropping of a connection, since the communication program would need to respond to such an event.

Broadcast consumers typically have a set of kinds of events in which they are interested and want to be notified of the next event from this set as simply as possible. Many broadcast consumers are interested in events that occur only to a limited set of subjects, and so one embodiment of the present invention provides a method for filtering events based on "who" they involve. In addition, the set of events in which the broadcast consumers are interested changes over time as does the immediacy of the interest. Thus, one embodiment of the present invention provides a method which allows the consumer to modify the set of events which will be delivered to the consumer. Lastly, there is no clear pattern as to whether broadcast consumers want to poll the event manager control unit to collect an event, or if they want to be notified asynchronously. Therefore, according to one embodiment, both options are supported.

FIG. **1** is a block diagram showing an exemplary computer on which the software according to the present invention can be implemented. The computer **10** includes a processor (CPU) **12**, an external storage device **14**, a disk storage device **16**, an input/output (I/O) controller **18**, an input device **20**, an output device **22**, a memory **24**, and a common system bus **26** connecting each of the above

elements. Only one input device and one output device is shown in FIG. 1 for ease of readability purposes. However, it will be appreciated that the computer **10** can include more than one such device. The processor **12**, the external storage device **14**, the disk storage device **16** and the memory **24** are also connected through the bus **26** and the I/O controller **18** to the input and output devices **20** and **22**. In addition, the computer **10** can also be adapted for communicating with a network **27**.

Stored within the memory **24** of the computer **10** are a number of pieces of software which can be executed by the processor **12**. One of those pieces of software is an operating system **28**. In one embodiment, the operating system **28** is a microkernel operating system capable of maintaining multiple address spaces. An event manager **30** resides within the operating system. In an exemplary embodiment, the computer system **10** of the present invention is an Apple Macintosh™ computer system made by Apple Computer, Inc., of Cupertino, Calif., and having a microprocessor and a memory wherein a microkernel operating system **28** that includes the event manager **30** resides. The components of the computer system can be changed within the skill of the ordinary artisan once in possession of the instant disclosure. Although the present invention is described in a Macintosh™ environment, it is within the scope of the invention, and within the skill of the ordinarily skilled artisan, to implement the invention in a DOS, Unix, or other computer environment.

The present invention relates to an architecture for an event manager for managing events that occur in an operating system. Clients of the event manager include event consumers and event producers, that is, applications programs and the various parts of the operating system, such as for example, a file manager.

According to one embodiment, the present invention cooperates with an operating system with a microkernel architecture in which the kernel provides a semaphore synchronization mechanism and a messaging system. The messaging system creates and maintains a set of message objects and one or more port objects. The messaging system has a number of features. It allows a creator-defined value to be associated with each object. The messaging system also allows multiple objects to be mapped to the same port and messages to be either received from a port or have a function be called when a message of an appropriate type is sent to that port. Further, the messaging system allows the receiver of the message to determine the object to which the message was originally sent and to derive the creator-defined value for that object. One example of such a microkernel architecture is provided by NuKERNEL™, used in Apple Macintosh™ computers. The NuKERNEL™ system is described in copending U.S. patent application Ser. No. 08/128,706, filed on Sep. 30, 1993, for a "System For Decentralizing Backing Store Control Of Virtual Memory In A Computer", application Ser. No. 08/220,043, filed on Mar. 30, 1994, for an "Object Oriented Message Passing System And Method", and an application entitled "System and Method Of Object Oriented Message Filtering", filed in the name of Thomas E. Saulpaugh and Steven J. Szymanski, on or about the date of filing of the present application. These three patent applications are incorporated by reference herein.

According to the messaging system, a service is provided by a server, and software wishing to make use of the service is called a client of the server. Messages are sent to a target by clients and received by servers. The message directs the server to perform a function on behalf of the client. Message objects are abstract entities that represent various resources to messaging system clients. These objects may represent, for example, devices, files, or windows managed by a server. Clients send messages to objects, which objects are identified by an identification labelled ObjectID.

Message ports are abstract entities that represent a service. These ports may represent, for example, a device driver, a file system, or a window manager. Servers receive messages from ports, which ports are identified by a label PortID. Objects are assigned to a port. A message sent to an object is received at that object's port. Ports and objects are created by the messaging system on behalf of a server. The creation of an object requires designating a port from which messages sent to the object will be retrieved.

According to one embodiment, the messaging system is used by the event manager in that the event distributors have an associated port and object, the sequential consumers have an associated port and object, and the event manager maintains a port and objects assigned to that port for each broadcast consumer. The events are passed as messages by the event producers to the distributor's associated object, the sequential consumers receive the messages from their associated port, and the broadcast consumers send messages to request events to the event manager via the object maintained for that broadcast consumer. Additionally, according to one embodiment, the event manager uses the message filter mechanism provided in a messaging system to implement sequential consumers. For each sequential consumer, a message filter is created on the distributor's ObjectID. The code which receives messages through the filter checks for matches (on the what and who provided) and forwards the appropriate messages to the sequential consumer. One example of such a message filtering mechanism is described in the above-referenced patent application in the name of Thomas E. Saulpaugh and Steven J. Szymanski.

FIG. **2** is a block diagram of the architecture for the event manager **30** shown in FIG. **1**. The event manager **30** (shown by the dashed line in FIG. **2**) is operationally connected to and communicates with a plurality of event distributors **340**, corresponding in number to the different kinds of events possible within the system, a plurality of event consumers, and a plurality of event producers **300**.

According to one embodiment, the plurality of event consumers can include broadcast consumers **310** and sequential consumers **360**. It is appreciated that a given system may have either one or more broadcast consumers or one or more sequential consumers, or both.

According to the present invention, events are grouped into "kinds", for example, all keystrokes are one kind of event, all mouse clicks are another kind of event, all new file creations am another kind of event, etc. It is also possible to group the events differently, for example, all depressions of a particular key are one kind of event. This is not a preferred implementation at least in part because the number of kinds of events would quickly become unmanageable. However, the choice of how to group the events is within the skill of the ordinary artisan once in possession of the present disclosure.

The event manager **30** includes an event manager control unit **305** and data structures. The data structures include a subscription matrix **330**, a sequential consumer database **350**, a plurality of event queues **320** provided in one-to-one correspondence with the broadcast consumers **310**, and a plurality of event kind headers **331** provided in one-to-one correspondence with the event distributors **340**. The event manager control unit **305** consists of at least one software routine which manages the event manager data structures.

Event producers **300** represent any software on a computer that is responsible for generating an event, or is responsible for detecting that other entities in the computer have generated an event. The event producers generate descriptions of each event they produce or detect. Each kind of event can have any number of event producers, and a given event produce may produce or detect multiple kinds of events. Event consumers, including broadcast consumers **310** and sequential consumers **360**, represent any program that needs to be informed when an event has occurred and needs to be informed of the description of that event. Each kind of event can have any number of event consumers, and a given event consumer may need to be notified of multiple kinds of events.

It is possible for an event consumer to be an event producer and for an event producer to be an event consumer. In particular, there are many cases where a piece of software will consume one kind of event, and produce another kind in response. For example, the file manager might consume "disk inserted" events, mount the volumes on the inserted media, and then produce "new volume" events as a result. It is also possible that a given program may subscribe to different kinds of events differently. In particular, the program may subscribe as a sequential consumer for some kinds of events and as a broadcast consumer for other kinds of events.

Event queues **320** are lists of events that are maintained by the event manager control unit **305** for each of the broadcast consumers **310** to hold information about events of interest to the corresponding broadcast consumer. According to one embodiment, the event queues **320** are structured as first-in, first-out lists. Other suitable storage configurations such as lists ordered by event priority or by producer priority, for example, disk inserted events may always have higher priority than mouse clicks or disk event producers may have higher priority than mouse event producers, may also be used. In FIG. **2**, event queue #1 is the list of events of interest to broadcast consumer #1, event queue #2 is the list of events of interest to broadcast consumer #2, and so on. Although four broadcast consumer/queue pairs are shown in FIG. **2**, it is appreciated that any number of broadcast consumer/queue pairs can be defined at a given time.

According to one embodiment, the event queues **320** are stored in the format shown in FIG. **3**, including the following fields: first subscription; last subscription; maximum events; next event; and event array. The first and last subscription fields are pointers to the first and last subscription for the corresponding broadcast consumer **310**. An event subscription is a description of a specific set of events of which a particular broadcast consumer needs to be informed. The maximum events field is the maximum number of events that are to be maintained in the queue for that consumer. According to one embodiment, the size of the event array may be slightly larger than the indicated maximum to provide a cushion. The event array stores the list of events that have occurred in the form of an array of elements, each element being an event to which the corresponding broadcast consumer has subscribed. In one embodiment, the queues are stored in circular buffers, although other suitable storage configurations such as singly and doubly linked lists and dynamic stacks could be used.

Subscription matrix **330** is a structure that maintains the information about all existing event subscriptions. In particular, the subscription matrix **330** is used to keep track of the subscriptions for the events in which the broadcast consumers are interested.

A plurality of sequential consumers **360** may be defined. Although three sequential consumers #1, #2, and #3, are shown in FIG. **2**, it is appreciated that any number of sequential consumers can be defined. The sequential consumer database **350** is composed of a plurality of sequential consumer entries which list the events in which each sequential consumer **360** is interested.

In one embodiment, the subscription matrix **330** can be configured as illustrated in FIG. **4**, where there is one subscription structures **3300** stored for each subscription. Other suitable configurations may be used such as sparse arrays, or dynamic lists of subscriptions on the event queue and/or event kind structures. The subscription structures **3300** stored in the subscription matrix are connected to the event queues **320** and the event kind headers **331**. In particular, there is one event kind header **331** stored for each kind of event known to the system, thus the number of event kind headers is equal to the number of event distributors. The event kind headers **331** store information for the event manager control unit **305** to use to determine which distributor handles the kind of event currently being processed. A RegisterDistributor (described below) call creates the event kind header.

According to one embodiment, the subscriptions **3300** are stored in the subscription matrix **330** in the format shown in FIG. **5***a*, including the following fields: event queue pointer; event kind header pointer; next subscription, same event; previous subscription, same event; next subscription, same queue; previous subscription, same queue; and event subject. The event queue pointer is a pointer to the event queue to which the subscription belongs. The event kind header pointer is the pointer to the event kind header for the kind of event which is the subject of the subscription. The event subject identifies the structure which the event happened to. The rest of the fields are the pointers to the other subscriptions.

An exemplary format for the sequential consumer entries is illustrated in FIG. **5***b*. According to one embodiment, the entries each include the following fields: event identifier, event subject and sequential consumer object ID. The event identifier identifies the kinds of events in which the sequential consumer is interested. The event subject identifies the subject of the events in which the sequential consumer is interested. The sequential consumer object ID identifies the objectID for the sequential consumer which created the entry in the sequential consumer database. According to one embodiment, the sequential consumer database **350** is part of the subscription matrix **330** and there is a pointer to the subscription structure for each sequential consumer. According to another embodiment shown in FIG. **2**, the sequential consumer database **350** is provided separately from the subscription matrix **330** as described above. The event manager control unit **305** performs a comparison between the data structure and the detected event to determine whether the event should go to that sequential consumer.

According to one embodiment, the message filtering mechanism is used to maintain the information required to provide the appropriate sequential consumers with the events as they occur. This mechanism is generally described in the Saulpaugh and Szymanski patent application discussed above.

According to one embodiment, the event kind headers **331** are stored in the format shown in FIG. **6**. The event kind header **331** includes the following fields: next header; DistributorObjectId; event identifier; first subscription; and last subscription. The next header field is a pointer to next header in a singly linked list of event kind headers. The DistributorObjectID is the distributor ObjectID for the correspond-

ing event distributor. The event identifier identifies the kind of event the associated distributor handles. The first and last subscription fields are pointers to first and last subscriptions defined for this kind of event.

Event distributors **340** are programs that are responsible for interpreting event subscriptions so as to distribute appropriate events to appropriate broadcast consumers. According to one embodiment, them is an event distributor for handling each kind of event. In particular, there is an event distributor to handle all keystrokes detected by any of the event producers, another event distributor to handle all mouse clicks, and so on.

According to one embodiment, it is up to the distributors to determine which broadcast consumers am notified of the event. The event manager control unit **305**, using the subscription matrix **330**, keeps track of which consumers want the events, while the distributors have the final say as to which consumers are actually notified. A given distributor might always give the result to all interested consumers, or it might always choose one, or it might ask to see each consumer and choose some and not others. This allows for a very flexible architecture. For example, while all applications will be interested in mouse clicks (and therefore would subscribe to that kind of event), only one (either the frontmost or the application for which a window is clicked on) should actually receive it. The mouse click event distributor determines which of the subscriptions was for the frontmost application and sends it only to that one. Another example would be that there might be several pieces of code which would want to know when a new file was created, and generally all of them should be told. Therefore, the new file event distributor would always send the event to all subscribers.

As shown in FIG. **2**, each broadcast consumer **310** communicates with and is operationally connected to its respective event queue **320** through the event manager control unit **305**. Each broadcast consumer **310** also communicates with and is operationally connected to the subscription matrix **330** through the event manager control unit **305**. The event queues **320** each communicate with and are operationally connected to the subscription matrix **330** through the event manager control unit **305**. The sequential consumer database **350** communicates with and is operationally connected to the event manager control unit **305** and the sequential consumers **360** communicate with the sequential consumer database **350** via the event manager control unit **305**. The plurality of event kind headers **331** are operationally connected to the plurality of event distributors **340**, which are operationally connected to the plurality of event producers **300**.

Each broadcast consumer **310** uses the event manager control unit **305** to cream an event queue **320** to hold events in which it is interested between the time the event is reported and the time the broadcast consumer consumes it. Each broadcast consumer **310** communicates to the event manager control unit **305** a set of event subscriptions that describe all the events of which it needs to be informed. This set of event subscriptions may be changed at any time. The event manager control unit **305** stores that information in the subscription matrix to be used by the event distributors **340**.

In summary, according to one embodiment of the present invention, event producers **300** detect events and build event descriptions. They send those descriptions to the event manager **30** by calling the event manager control unit **305**. The event manager control unit **305** sends the event description to each sequential consumer **360** in turn based on the entries in the sequential consumer database **350**. The event manager control unit **305** then sends the event description to the event distributor **340** who is responsible for distributing that kind of event. The event distributor **340** calls the event manager control unit **305** to send the event description to those broadcast consumers **310** which it decides are appropriate based on the information in the subscription matrix **330**. The event manager control unit **305** gives the event descriptions to the appropriate broadcast consumers **310** by initially storing those descriptions in the event queues **320**. The broadcast consumers **310**, when ready, call the event manager control unit **305** to retrieve the next event description stored in its corresponding event queue **320**.

## APPLICATION PROGRAMMER INTERFACE

According to the present invention, the event consumers, the event producers and the event distributors may be written by third parties other than the manufacturer of the event manager. Therefore, an application programmer interface (API) is defined to provide a specification which allows these third parties to communicate with the event manager. The following is a description of one embodiment of an API which allows communication with the event manager according to the present invention.

### Event Structure

According to one embodiment, an event is composed of three parts: what happened (the event identifier), who it happened to (the event subject), and details of how the event happened (the event information). Since it is desirable for the set of possible events to be easily extensible (by the developers of the other parts of the operating system and by applications programmers), the "what" part of the event structure is defined to be a unique identifier referred to as the event name. In one embodiment, the event name identifier can be implemented as a 4 character code known as an OSType. It is appreciated that other suitable identifiers could be used instead by one of ordinary skill once in possession of the present disclosure. According to one embodiment, it can be a pair of identifiers. In particular, in this disclosure, the implementation described uses two OSTypes. An exemplary structure **45** for the event name identifier is shown in FIG. **7**. The event service identifier serves as the "signature" or name of the service which defined the event, e.g., a word processing program, and the event kind identifier identifies the event itself, e.g., new file created. Thus, the universe of all names, referred to as the namespace, of events is managed by controlling the signatures.

The "who" field of the event structure, defined according to one embodiment as the event subject, is difficult to define since all possible uses of the event manager can not be anticipated and therefore all forms of the hardware and software elements that the "who" might describe also can not be anticipated. Fortunately, the only public operation which needs to be supported for this part of an event is a test for equality, i.e., equality against a "who" which was provided by a broadcast consumer for a subscription, so the event manager control unit **305** need not know the structure of the subscription and it can be defined as an uninterpreted array of bytes. Suitable configurations of the "who" field can be used, such as a fixed length or a variable length field, within the skill of the ordinary artisan once in possession of the present disclosure.

The "details" part of the structure, that is, the info field in FIG. 8, is totally open ended, and will vary not only with each event, but potentially with each instance of the event. Thus, one embodiment for the structure is an open ended array of bytes.

FIG. 8 is a block diagram of the event structure according to one embodiment of the present invention. The event 40 consists of the event name, event subject, byte count, and event info fields. The event name field specifies the what, the event subject field specifies the who, and the byte count and event into fields specify the how. Event info is the actual data and byte count indicates the length of the data.

## System Calls

The event producers are provided with a single call to submit events to the event manager control unit 305. In particular, the call sends the message of an event to the event manager control unit 305. According to one embodiment, the producers then receive confirmation that every consumer who needs to respond to the event has had a chance to process it. In particular, when the call returns, the distributor knows that every sequential consumer has seen it, and it is enqueued for every broadcast consumer. This confirmation is not required and can be omitted according to another embodiment. According to one embodiment, this call has been split into two calls (FindDistributor and ProduceEvent) For efficiency reasons to actor out the identification of who knows how to distribute that kind of event. The API is shown in Table 1.

Table 1 and the succeeding tables present structure definitions written in the C language. While the examples herein are shown in C, it is within the skill of the ordinary artisan to use other suitable programming languages to implement the present invention. In addition, the code shown in the Tables is an example of an implementation of the invention according to one embodiment. It is appreciated that other implementations are possible and within the skill of the ordinary artisan once in possession of the present disclosure.

In Table 1, OSStatus is a type usually used as a return value used to indicate if the call succeeded. OptionBits is a type usually used as an input parameter to a call to allow the user to specify small variations on how the call is to be processed.

TABLE 1

```
typedef Object ID                          EventDistributorID;
typedef OptionBits                         ProduceEventOptions;
enum
    {
    kProduceEventsynchronously             = 0x00000001
    // don't return until all Sequential
    Consumers have completed
    }
OSStatus FindDistributor (      EventName              eventname,
                                EventDistributorID     *inputID);
OSStatus ProduceEvent (         EventInformationPtr    event,
                                EventDistributorID     distributor
                                ProduceEventOptions    options);
```

## Consumers

According to one embodiment of the present invention, as discussed above, broadcast consumers and sequential consumers differ in their relationships to other consumers of an event. Broadcast consumers do not need to know if other consumers exist, nor in what order consumers are informed of the event, as long as they themselves are eventually informed. Sequential consumers require that no other consumer be told about an event while they themselves are still processing it, and they require the ability to influence when in the sequence they receive the event. In addition, many sequential consumers require the ability to modify the event itself, and even block an event from being received by other consumers.

According to one embodiment, the API provides a method for filtering events based on "who" they involve to allow broadcast consumers to limit the set of subjects for which they will be notified about events. In addition, the API provides a method which allows the broadcast consumer to modify the set of events in which it is interested over time. Lastly, the API supports the ability of broadcast consumers to poll the event manager control unit 305 to collect an event, or to be notified asynchronously by use of a messaging system providing asynchronous notification functionality.

In the interface according to one embodiment, the consumer calls the event manager control unit 305 to create a consumer structure which embodies a list of events in which the consumer is interested. These structures are opaque to the caller (i.e., its details are known only to the event manager control unit 305).

According to one embodiment, this consumer structure is an event queue 320. The API allows the broadcast consumers to call the event manager control unit 305 to create an event queue. In particular, the event queues 320 are created by the event manager control unit 305 as a result of a CreateBroadcastConsumer call, and they are eliminated as a result of a DisposeBroadcastConsumer call. One embodiment of the details of the queues are illustrated in FIG. 3. The event queue includes, among other things, the name of the consumer (used by distributors to identity specific consumers), and the maximum number of events the event manager control unit 305 will buffer by storing in the event queue for that consumer. The consumer is asked to provide this latter value so the event manager control unit 305 is not committed to an unbounded amount of buffering. Should this value be exceeded, additional events are discarded and the next event the consumer will get is an "Overrun" event from the event manager control unit 305 itself. The example code in Table 2 illustrates this feature.

TABLE 2

```
dypedef ObjectID EventConsumerID;
typedef FilterName EventConsumerName;
OSStatus CreateBroadcastConsumer (
    EventConsumerID                *consumer,
    EventConsumerName              name,
    uint32                         maxPending);
OSStatus DisposeBroadcastConsumer (
    EventConsumerID                consumer);
```

Once the broadcast consumer has an EventConsumerID, it subscribes to the kinds of events in which it is interested. As part of the subscription process the broadcast consumer can specify both the EventName and the EventSubject to be matched for determining when the consumer needs to be notified of the event. The policies of how the EventSubject is matched by the event distributor 340 is dependent on the kind of event being processed. According to one embodiment, this can be implemented as a byte-by-byte comparison. Other suitable implementations may be used, for example, if there are three different kinds of events that are identical for subscription purposes, i.e., the same consumers are to be notified of them, therefore, these events are

compared as equal even though the actual bits are different. The list of events associated with an EventConsumerID can be expanded at any time using SubscribeBroadcastConsumer, and the events kinds can be removed from the list at any time with UnsubscribeBroadcastConsumer. If a particular kind of event is unsubscribed, any event instances of that kind which have already been collected for that consumer are discarded. This is illustrated in Table 3.

TABLE 3

```
OSStatus SubscribeBroadcastConsumer (
    EventConsumerID                 consumer,
    EventName                       eventName,
    EventSubjectPtr                 subject);
OSStatus UnsubscribeBroadcastConsumer (
    EventConsumerID                 consumer,
    EventName                       eventName,
    EventSubjectPtr                 theSubject);
```

Processing Events

The HoldEvents/UnholdEvents calls are provided to simplify some special cases in the processing of events. When a kind of event is held, instances of those events are still collected for the broadcast consumer; however, they will not be returned to the user by a Consume call until they are unheld. In this way, the event manager handles situations in which particular kinds of events are still of interest to a broadcast consumer, but the processing of those events are temporarily impossible or not a priority. According to another embodiment, the same functionality could be provided by creating two separate broadcast consumer IDs, one for the non-held events and one for the holdable events. The broadcast consumer could then choose when to consume from the second EventConsumerID. However, this embodiment may not be completely practical because it is not always possible to anticipate what will belong in which category, and the decision process for holding is often quite removed from the event processing cycle. The HoldEvents/UnholdEvents calls, according to one embodiment, provide a simple interface for separating that decision process. These calls are illustrated in Table 4.

TABLE 4

```
OSStatus HoldEvents ( EventConsumerID       consumer,
    EventName                               eventName,
    EventSubjectPtr                         subject);
OSStatus UnholdEvents ( EventConsumerID     consumer,
    EventName                               eventName,
    EventSubjectPtr                         subject);
```

Finally, when the list of interesting events is built, the broadcast consumer consumers the events which are collected using one of two calls, a synchronous call and an asynchronous call. Both calls specify the consumer objectID to identify the queue, and provide a buffer into which the event is copied. The Async call additionally requires a EventNotification structure which describes the action to be performed when a buffer has been filled in with a new event. According to one embodiment, the EventNotification and EventInformationPtr are types defined by an operating system kernel. If the buffer provided is not large enough to hold the event description, the leading part of the description is copied into the buffer. The rest of the description is dropped. These calls are illustrated in Table 5.

TABLE 5

```
OSStatus ConsumeEvent (
    EventConsumerID                 conswner,
    uint32                          maxEventSize,
    EventInformationPtr             event);
OSStatus ConsumeEventASync (
    EventConsumerID                 conswner
    EventNotification               *completion,
    uint32                          maxEventSize,
    EventInformationPtr             event);
```

FlushEvents is used to remove events which have been collected for the broadcast consumers but which have not been consumed. The broadcast consumer specifies an EventName (which can use wildcards) and EventSubject. The event manager control unit **305** then disposes of any events which match the description and were collected but not processed. This call is illustrated in Table 6.

TABLE 6

```
OSStatus FlushEvents (
    EventConsumerID                 consumer,
    EventName                       eventname,
    EventSubjectPtr                 subject);
```

Sequential Consumers have very different requirements tier their API. Since they serve as a bottleneck for the transmission of events, their API should be designed for maximum throughput; and should be structured to require a response for each event to indicate when the event manager control unit **305** can pass the event on to the next sequential consumer or the broadcast consumers. Based on this, the API according to one embodiment of the present invention assumes that events are passed to sequential consumers as messages using a suitable communication facility, such as the messaging system service provided by a microkernel operating system as discussed above. According to one embodiment, the sequential consumer sends a message to the event manager control unit **305** asking for the next event and the event itself is returned in the reply to the message. Other implementations are also possible within the skill of the ordinary artisan once in possession of the present disclosure. According to one embodiment, sequential consumers receive these messages by way of an accept function or by asynchronous receives to allow the receiver to overlap program execution with the receive operation to insure a rapid response time. One example of such a messaging system is provided by NuKERNEL™, discussed above.

According to one embodiment, since the messaging system provides a funneling mechanism, having multiple objects associated with a single port, no subscription calls are needed for sequential consumers. Instead, the sequential consumer will be asked to create an object for each interesting kind of event, and those objects can be associated with ports in any way the consumer desires. The objects are then associated with a kind of event using the API described below. Other suitable embodiments of this feature are within the skill of the ordinary artisan once in possession of the present disclosure.

According to one embodiment, the actual sequencing of the consumers is provided by using the message filtering mechanism to transparently list, screen, alter, or re-route messages. Message filters are a software fabricated message interception device which can be ordered in a predetermined priority order to allow certain filters to take precedence over others. One embodiment of the present invention implements the sequencing of the consumers using this ordering

capability. This mechanism is described in detail in the copending application filed in the name of Saulpaugh and Szymanski referenced above.

The InstallSequentialConsumer call is used to tell the event manager control unit **305** that events of the given name and subject should be passed to the given object. The name, ordering and options parameters are used to determine the exact ordering in which those events are given to the sequential consumers that are interested in them. The RemoveSequentialConsumer call is used to remove the installed sequential consumer. The installation and removal calls are illustrated in Table 7.

TABLE 7

```
typedef FilterID SequentialConsumerID;
typedef FilterName
SequentialConsumerName;
OSStatus InstallSequentialConsumer (
    SequentialConsumerID          *consumer,
    SequentialConsumerName        consumersName,
    ObjectID                      consumersObject
    FilterOptions                 consumerOptions,
    FilterobjectPair              consumerOrdering,
    EventName                     eventname,
    EventSubjectPtr               eventSubject);
OSStatus RemoveSequentialConsumer (
    SequentialConsumerID          consumer);
```

Once a sequential consumer has been installed and has received or consumed an event, the sequential consumer needs to indicate when it is done with the event so the event manager control unit **305** can pass it on to the next sequential consumer (or to the broadcast consumers if there am no more sequential ones). In particular, the sequential consumer makes a NextConsumer call to the event manager control unit **305** indicating that it has completed processing of the event. The event manager control unit **305** then passes the event to the next sequential consumer or to the appropriate broadcast consumers. Instead of passing the event on, a sequential consumer has the ability to declare an event handled so that it will not be distributed to any more consumers of any type. In particular, a sequential consumer can make an EventHandled call to prohibit further distribution of the event. These two calls are illustrated in Table 8. In one embodiment, NextConsumer uses the messaging system call ContinueMessage which acts as an automatic forward command to continue to the next object in a filter chain in the messaging system, and EventHandled uses the messaging system call ReplyToMessage which specifies that all the objects in a chain have processed the message. This mechanism is described in detail in the copending application filed in the name of Saulpaugh and Szymanski referenced above.

TABLE 8

```
OSStatus NextConsumer (     MessageID     eventMessageID);
OSStatus EventHandled (     MessageID     eventMessageID);
```

## Distributors

As discussed above, each distributor is responsible for processing different kinds of events. Each distributor must therefore know the events for which it is responsible.

According to one embodiment, most events are routed through the appropriate sequential consumers and then copies are given to all of the broadcast consumers who have expressed interest. According to a default implementation, any matching of event subjects by the event distributors is done by direct byte comparison of the whole value to determine which consumers have to be notified of the event. Other suitable implementations are within the skill of the ordinary artisan once in possession of the present disclosure.

According to one embodiment, a default distributor is provided to pass certain kinds of events to all broadcast consumers. For example, events such as battery low events will always be distributed to all interested consumers. Each event that can be distributed by the default distributor must register with the default distributor by the RegisterEventWithDefaultDistributor call, illustrated in Table 9. This call is made one time per kind of event over the life of the system to tell the default distributor that it is responsible for those events.

TABLE 9

```
OSStatus RegisterEventWithDefaultDistributor(
    EventName          eventName);
```

A custom distributor is registered with the event manager control unit **305** for handling each kind of event that is not handled by the default distributor. According to one embodiment, registering a custom distributor with the event manager control unit **305** is performed by a call which associates an event with an object so that when the associated event occurs, the event manager control unit **305** knows which distributor will handle the distribution of that event. The custom distributors are independently loaded services and all communication between producers and distributors is accomplished by sending and receiving messages. To receive the events to process and distribute, each distributor must provide an ObjectID to which events are sent from the producers. Calls for registering and unregistering custom distributors are illustrated in Table 10.

TABLE 10

```
OSStatus RegisterDistributor(
    EventName          eventname,
    ObjectID           distributor);
OSStatus UnregisterDistributor(
    EventName          eventname,
    ObjectID           distributor);
```

There are several cases where the distributor needs to apply more complicated heuristics to the process of determining which broadcast consumers should receive notification of events reported by the producers. The information in the subscription matrix is available to the distributor to help determine which broadcast consumers to notify. According to one embodiment, a distributor cannot give an event to a broadcast consumer who has not asked for it, and the distributor looks in the subscription matrix to find out who asked for it. However, distributors are allowed to give the event to only a subset of the broadcast consumers who asked for it. The kinds of ways a distributor might need to effect the distribution of events include:

1. The way in which event subjects are matched is more complicated than direct byte comparison. There may be substructure in the subject identification which needs to be taken into account, or there may be some form of wildcard processing which needs to be handled.
2. Particular events may need to be routed to a subset of the broadcast consumers who have expressed interest, perhaps only one. For instance, mouse events need to be routed to only the one appropriate application or those applications which are interested in them.

3. Related events may need to be combined. For instance, multiple update region events for the same window should be combined until the application actually consumes the event.

Note that none of these possibilities affect sequential consumers. This is because one of the reasons sequential consumers are needed is to influence the distribution process. In other words, sequential consumers are so labelled because for the kinds of events in which they are interested, they must react to the event. Thus, there are no circumstances under which a distributor would choose not to send the event to a particular sequential consumer. Therefore, the event manager control unit **305** automatically sends all produced events through the gauntlet of sequential consumers before handing them off to the distributors.

FIGS. **9A**, **9B**, and **9C** are flowcharts illustrating the event handling process from the event producers' and the event distributors' point of view according to one embodiment of the present invention.

As shown in FIG. **9A**, the event producer first detects or generates an event (step **600**). The event producer generates a description of the event (step **602**) and calls the event manager control unit **305** to send the event to the appropriate event consumers, both broadcast and sequential consumers (step **604**).

FIG. **9B** illustrates the processing of the event by the event manger control unit **305** according to one embodiment of the present invention. It is appreciated that although the following description assumes the use of a messaging system to send the events between elements of the computer, suitable alternative embodiments can be implemented by an ordinarily skilled artisan once in possession of the present disclosure.

At step **606**, a determination is made whether the event producer needs to know when the event has been handled, that is, whether to process the request asynchronously or synchronously. If the event producer does not need to know when the event has been handled (as indicated by an input parameter to the call), it returns to the producer (step **608**) and processes the rest of the steps in this Figure asynchronously (step **610**). If the event producer does need to know, the steps are processed synchronously (step **612**).

The test is implemented by making the message sent in step **606** an asynchronous message, otherwise it is a synchronous message. In particular, the event producer calls the event manager control unit **305** to send a message concerning the detected event. The producer provides an input parameter having a value which indicates whether it wants to wait for a response, in which case the message is sent synchronously. Otherwise, the message is sent asynchronously. By definition, if the message is sent synchronously, the event manager control unit **305** does not return from that call until the message has been processed by all sequential consumers; if the message is sent asynchronously, as soon as the message is sent, the event manager control unit **305** returns back to the event producer so that the event producer does not know whether the message is processed.

At step **614**, the event manager control unit **305** sends the event to each sequential consumer having an entry in the sequential consumer database **350** matching the event description. This is done one at a time, and each sequential consumer can, by calling the event manager control unit **305**, either have the message sent to the next sequential consumer or declare that the event has been handled. Thus, after the sequential consumer completes processing the event, the sequential consumer indicates whether processing of the event should continue or whether the event processing should stop because the event has been handled (step **616**). A determination is then made by the event manager control unit **305** whether that sequential consumer indicated that the event was handled (step **618**). If so, the event is prohibited from being distributed to any other consumers and so the routine returns to the calling producer. If the sequential consumer indicated that the event was not handled, a determination is made whether that sequential consumer is the last one (step **622**). If not, the distributor returns to step **616** to distribute the event to the next sequential consumer.

Referring now to FIG. **9C**, when the event has been distributed to all sequential consumers (step **622**), the event manager control unit **305** sends the event to the event distributor responsible for that kind of event to distribute to the broadcast consumers (process **624**). The event is sent as a message to the distributor. The messaging system automatically passes it on to the distributor when all of the filters have processed it and none have replied (thereby declaring the event handled). Other suitable implementations are possible, such as having the event manager control unit **305** call the event distributor at this point, are within the skill of the ordinary artisan once in possession of the present disclosure.

Process **624** is illustrated in FIG. **9D**. The distributor decides how to distribute the event in this process. According to one embodiment, it can do this by calling the event manager control unit **305** to sequence through all of the subscriptions for that kind of event to see, which of the consumers who have subscribed should get the event.

One situation is where the distributor needs to route an event to all or some of the consumers; but needs to be able to modify the event data itself up until the point at which the consumer receives and consumes it (step **6240**). For example, in a windowing environment, when a window is uncovered (i.e., the topmost window is closed), a window uncovered event is generated. Because it is possible that another portion of the uncovered window may still be uncovered, the window uncovered event distributor wants to be able to modify this event up until the point at which it is consumed. In this case, the distributor submits a place holder structure for the actual message, and gives it to the appropriate consumers.

The place holder structure includes a reference constant field, RefCon, to be used by the distributor to identify the event, and an ObjectID to be used to get the actual data from the distributor. The distributor provides the value of the RefCon field when it creates the placeholder, and is responsible for being able to take that value and provide back the actual event description. Thus, when the consumer actually tries to consume the placeholder, the event manager control unit **305** sends a message to the given ObjectID containing the placeholder RefCon. The distributor then fills in the buffer with the actual, current event description and the consumer's buffer is passed to Consume. Since this functionality needs to be provided for both select and all consumers, two calls are provided as shown in Table 11. If all consumers are to be notified of an event using a place holder (step **6248**), the GivePlaceholderToAllConsumers call is used (step **6250**). If only select consumers are to be notified of an event using a place holder (step **6248**), the GivePlaceholderToSelectConsumers call is used (step **6252**).

TABLE 11

```
OSStatus GivePlaceholderToAllConsumers
    EventInformationPtr          event,
    void*                        placeholderRefCon,
    ObjectID                     fulfillmentID);
```

TABLE 11-continued

```
OSStatus
GivePlaceholderToSelectConsumers
    EventInformationPtr          event,
    ConsumerFilterFunction       filterFunc,
    void*                        placeholderRefCon,
    ObjectID                     fullfilemntID);
```

If the distributor does not need to modify the event data, the result of the determination at step **6240** is no. Then, in step **6242**, a determination is made if all of the broadcast consumers should receive the event. Note that if this is the case all of the time, it is preferable to use the default distributor which automatically notifies all consumers. Alternatively, if special processing is required only for a subset of events, for any event not in that subset, the GiveEventToAllConsumers call can be used to invoke the distribution mechanism (step **6244**). This call is illustrated in Table 12.

TABLE 12

```
OSStatus
GiveEventToAllConsumers(EventInformationPtr    event);
```

The situation in which the distributor needs to choose a subset of the consumers to receive the message may occur either because the distributor needs to process any subject matches or because the nature of the event requires limiting the distribution is detected by a no response to the test in step **6240**. In particular, in cases where a byte-by-byte comparison of the subject field is not desired, the distributor is responsible for determining if the subjects match. To accomplish this, the GiveEventToSelectConsumers call is used (step **6246**), and a filter function, ConsumerFilterFunction, is called for each consumer who has subscribed to that kind of event (regardless of the subject they specified). The function returns true if the event should be given to that consumer, false if not. This call is illustrated in Table 13.

TABLE 13

```
typedef Boolean (*ConsumerFilterFunction)(
    EventInformationPtr          event,
    EventConsumerName            consumer,
    EventSubjectPtr              subsciptnSubj);
OSStatus GiveEventToSelectConsumers(
    EvenInformationPtr           event,
    ConsumerFilterFunction       filterFunc);
```

After each of the calls at steps **6244**, **6246**, **6250**,and **6252**, the routine returns to the event manager control unit **305** (step **6254**).

Referring now to FIG. **9C**, at step **625**, the event manager control unit **305** enqueues the event according to which call was used in FIG. **9D**. According to one embodiment, the events are enqueued and dequeued in a first in, first out (FIFO) order. When an event consumer is ready to act on an event, it dequeues the top most event in the event queue and handles it as required.

If the steps of FIG. **9B** were processes synchronously (step **626**), the routine returns to the producer at step **628**. Otherwise, the routine replies to the producer's message (step **630**).

FIG. **10** is a flowchart illustrating the event handling process from the sequential consumers' point of view according to one embodiment of the present invention. First, the sequential consumer calls the event manager control unit **305** to get installed (step **700**). According to one embodiment, the consumer provides an ObjectID to which events are to be sent as messages. It is appreciated that other suitable implementations are possible, for example, the consumer could provide a function pointer which is called with a pointer to the event description. Such suitable implementations are within the skill of the ordinary artisan once in possession of the present disclosure.

The sequential consumer does whatever it wants while waiting for an event to occur (step **702**). According to one embodiment, since events are sent as messages, the sequential consumer calls the messaging system to either accept or receive a message through the given Object. Other things may be done by the sequential consumer, depending on whether the process is synchronous or asynchronous.

When an event is received (step **704**), the sequential consumer acts on the event in the appropriate way (step **706**). The sequential consumer then decides whether it wants other consumers (sequential or broadcast) to see this event or if it has handled the event well enough that others should not see it (step **707**). If the event is done, and others should not see it, an event handled message is sent by calling the event manager control unit **305** (step **711**). Otherwise, the sequential consumer calls the event manager control unit **305** to say that the event should be passed to the next consumer (step **709**). The sequential consumer then loops back to the wait state of step **702**.

FIG. **11** is a flowchart illustrating the event handling process from the broadcast consumers' point of view according to one embodiment of the present invention. The broadcast consumer first calls the event manager control unit **305** to register itself using the CreateConsumer call. Once registered, the distributor will start receiving messages through the object referred to by the given ObjectID. The contents of these messages are the descriptions generated by the producers of the events they have detected or generated. The distributor must then indicate to the event manager control unit **305** which of the currently executing consumers should receive the event. In one embodiment, the messaging system routes events from the producers, through the sequential consumers, and into the distributors. The distributor then uses the event manager control unit **305** to put the event on the appropriate queues.

The event manager control unit **305** creates an event queue for the consumer and returns (step **802**). The broadcast consumer then calls the event manager control unit **305** to subscribe to particular kinds of events in which it is interested (step **804**). The event manager control unit **305** adds the entries in the subscription matrix to keep track of the consumer's subscriptions (step **806**).

The consumer calls the event manager control unit **305** to get the next event (step **808**). According to one embodiment, this is implemented by having the consumer send a message to the event manager control unit **305**. The reply to that message will be the next event for that consumer. The consumer has the option of doing this synchronously or asynchronously, and the consumer either sends the message synchronously or asynchronously accordingly. Other suitable implementations are possible within the skill of the ordinary artisan once in possession of the present disclosure. The consumer acts on the event as it sees fit (step **810**), and the process loops back to step **808**.

The foregoing description of the specific embodiments will so fully reveal the general nature of the invention that others can, by applying current knowledge, readily modify and/or adapt for various applications such specific embodiments without departing from the generic concept, and, therefore, such adaptations and modifications should and are

intended to be comprehended within the meaning and range of equivalents of the disclosed embodiments. It is to be understood that the phraseology of terminology employed herein is for the purpose of description and not of limitation.

What is claimed is:

1. In a computer including at least one event producer for detecting that an event has occurred in the computer and generating an event and at least one event consumer which needs to be informed when events occur in the computer, a system for distributing events comprising:

storing means for storing a specific set of events of which said at least one event consumer is to be informed;

event manager control means for receiving the event from the event producer, comparing the received event to the stored set of events, and distributing an appropriate event to an appropriate event consumer; and

distributor means for receiving the event from the control means and directing said control means to distribute an appropriate event to an appropriate event consumer.

2. The system according to claim 1, wherein said distributor means comprises a distributor module for each kind of event possible in the computer.

3. The system according to claim 1, wherein a plurality of event consumers are included in the computer and the plurality of consumers comprise:

broadcast consumers having no relationship with other consumers, the broadcast consumers operating independently of other consumers and of the order in which consumers are informed of the event; and

sequential consumers having relationships with other consumers, the sequential consumers requiring that no other consumer be told about an event while they themselves are processing the event and having an ability to influence when they receive the event relative to the other consumers.

4. The system according to claim 3, wherein said distributor means comprises:

means for determining if the event is to be sent to broadcast consumers with a signal indicating that the distributor means maintains a right to modify the event based on subsequent events until the broadcast consumers receive the event; and

means, responsive to a positive determination by said means for determining, for directing said control means to distribute the event to the appropriate broadcast consumers with a signal indicating that the distributor means maintains the right to modify the event.

5. The system according to claim 3, wherein said distributor means comprises:

first means for determining if the event is to be sent to broadcast consumers with a signal indicating that the distributor means maintains a right to modify the event;

second means, responsive to a positive determination by said first means for determining, for determining if the event is to be passed to all broadcast consumers;

means, responsive to a positive determination by said second means for determining, for directing said control means to distribute the event to all broadcast consumers with a signal indicating that the distributor means maintains the right to modify the event;

means, responsive to a negative determination by said second means for determining, for directing said control means to distribute the event to select broadcast consumers with a signal indicating that the distributor means maintains the right to modify the event;

third means, responsive to a negative determination by said first means for determining, for determining if the event is to be sent to all broadcast consumers;

means, responsive to a positive determination by said third means for determining, for directing said control means to distribute the event to all broadcast consumers; and

means, responsive to a negative determination by said third means for determining, for directing said control means to distribute the event to select broadcast consumers.

6. The system according to claim 3, wherein said storing means comprises:

a subscription matrix for storing subscriptions to events in which the broadcast consumers are interested; and

a sequential consumer database for storing entries to events in which the sequential consumers are interested.

7. The system according to claim 3, wherein said storing means comprises an event queue corresponding to each of the broadcast consumers for receiving distributed events from said control means and for storing the distributed events until the events are consumed by the corresponding broadcast consumer.

8. The system according to claim 3, wherein said control means comprises means for passing an event to the sequential consumers in succession in accordance with the entries in the sequential consumer database.

9. The system according to claim 8, wherein said control means comprises means for prohibiting passing of an event upon receiving an event handled message from a sequential consumer.

10. A computer system comprising:

event producers for detecting that an event has occurred in the computer, generating an event, and generating a description of the event;

event consumers which need to be informed when events occur in the computer, said event consumers comprising a first and a second class of consumers;

storing means for storing a specific set of events of which the event consumers are to be informed;

event manager control means for receiving the event from the event producers and comparing the received event to the stored set of events;

distributor means, responsive to said event control means, for deciding if an event should be passed to an event consumer;

said event manager control means comprising:

first means for sending an event to appropriate event consumers of a first class in accordance with the stored set of events, and

second means for sending the event to appropriate event consumers of a second class responsive to said distributor means.

11. The system according to claim 10, wherein said distributor means comprises a distributor module for each kind of event possible in the computer.

12. The system according to claim 10, wherein

said first class of consumers comprise sequential consumers having relationships with other consumers, the sequential consumers requiring that no other consumer be told about an event while they themselves are processing it, and having an ability to influence when they receive the event relative to the other consumers; and

said second class of consumers comprise broadcast consumers having no relationship with other consumers, the broadcast consumers operating independently of other consumers and of the order in which consumers are informed of the event.

13. The system according to claim 12, wherein said distributor means comprises:

means for determining if the event is to be sent to broadcast consumers with a signal indicating that the distributor means maintains a right to modify the event based on subsequent events until the broadcast consumers receive the event; and

means, responsive to a positive determination by said means for determining, for directing said control means to distribute the event to the appropriate broadcast consumers with a signal indicating that the distributor means maintains the right to modify the event.

14. The system according to claim 12, wherein said storing means comprises:

a subscription matrix for storing subscriptions to events in which the broadcast consumers are interested; and

a sequential consumer database for storing entries to events in which the sequential consumers are interested.

15. The system according to claim 12, wherein said storing means comprises an event queue corresponding to each of the broadcast consumers for receiving distributed events from said control means and for storing the distributed events until the events are consumed by the corresponding broadcast consumer.

16. The system according to claim 12, wherein said control means comprises means for passing an event to the sequential consumers in succession in accordance with the entries in a sequential consumer database.

17. The system according to claim 16, wherein said control means comprises means for prohibiting passing of an event upon receiving an event handled message from a sequential consumer.

18. A method for distributing events occurring in a computer, said method comprising the steps of:

determining that an event has been detected by an event producer in the computer;

storing, in a storing means, a specific set of events of which an event consumer is to be informed;

receiving the event in an event control means from the event producer;

comparing the received event to the stored set of events;

receiving the event in a distributor means from the control means;

directing the control means to distribute an appropriate event to an appropriate event consumer; and

distributing, via the control means, an appropriate event to an appropriate event consumer.

19. The method according to claim 18, wherein the event consumer comprises a plurality of consumers including broadcast consumers which operate independently from one another and of the order in which consumers are informed of events and sequential consumers which require that no other consumer be told about an event while they themselves are processing it and have an ability to influence when they receive the event relative to the other consumers.

20. The method according to claim 19, wherein said step of distributing comprises the steps of:

determining if the event is to be sent to broadcast consumers with a right to modify the event based on subsequent events until the broadcasts consumers receive the event; and

distributing the event to the appropriate broadcast consumers with the right to modify responsive to a positive determination by said step determining.

21. The method according to claim 19, wherein said step of storing comprises the steps of:

storing, in a subscription matrix, subscriptions to events in which the broadcast consumers am interested; and

storing, in a sequential consumer database, entries to events in which the sequential consumers are interested.

22. The method according to claim 19, further comprising the steps of:

receiving distributed events in an event queue corresponding to a broadcast consumer; and

storing the distributed events in the event queue until the events are consumed by the corresponding broadcast consumer.

23. The method according to claim 19, wherein the step of distributing comprises the step of passing an event to the sequential consumers in succession upon receiving a continue message from a sequential consumer indicating that it has completed processing of the event.

24. The method according to claim 23, wherein the step of distributing further comprises the step of prohibiting passing of an event upon receiving an event handled message from a sequential consumer.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

# CERTIFICATE OF CORRECTION

PATENT NO. : 5,566,337
DATED : October 15, 1996
INVENTOR(S) : Steven J. SZYMANSKI, et al.

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

Item [57]

Replace the Abstract as follows:

In a computer including event producers for generating events and detecting that an event has occurred in the computer and event consumers which need to be informed when events occur in the computer, a system distributes information about the events. The system includes a store for storing a specific set of events of which an event consumer is to be informed, an event manager control unit for receiving the event from an event producer, comparing the received event to the stored set of events, and distributing the appropriate event to the appropriate event consumer, and a distributor for receiving the event from the control unit and directing the control unit to distribute the appropriate event to the appropriate event consumer. The system manages events within the computer by facilitating communication between the event producers and the event consumers without requiring each event producer to be aware of all of the event consumers.

**Signed and Sealed this**

**Twenty-eighth Day of January, 1997**

*Attest:* Bruce Lehman

**BRUCE LEHMAN**

*Attesting Officer* *Commissioner of Patents and Trademarks*

# EXHIBIT E

US00RE39486E

(19) **United States**

(12) **Reissued Patent**

**Cleron et al.**

(10) **Patent Number: US RE39,486 E**

(45) **Date of Reissued Patent: Feb. 6, 2007**

(54) **EXTENSIBLE, REPLACEABLE NETWORK COMPONENT SYSTEM**

(75) Inventors: **Michael A. Cleron**, Menlo Park, CA (US); **Stephen Fisher**, Menlo Park, CA (US); **Timo Bruck**, Mountain View, CA (US)

(73) Assignee: **Apple Computer, Inc.**, Cupertino, CA (US)

(21) Appl. No.: **10/408,789**

(22) Filed: **Apr. 3, 2003**

(Under 37 CFR 1.47)

**Related U.S. Patent Documents**

Reissue of:

(64) Patent No.: **6,212,575**
Issued: **Apr. 3, 2001**
Appl. No.: **08/435,377**
Filed: **May 5, 1995**

(51) **Int. Cl.**
***G06F 9/00*** (2006.01)
***G06F 9/46*** (2006.01)

(52) **U.S. Cl.** ..... **719/328**; 719/329; 709/201; 709/202; 709/203

(58) **Field of Classification Search** ..... 719/328–329; 709/200–203
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,297,249 A * 3/1994 Bernstein et al.
5,339,430 A * 8/1994 Lundin et al.
5,481,666 A * 1/1996 Nguyen et al.
5,530,852 A * 6/1996 Meske, Jr. et al.
5,537,526 A * 7/1996 Anderson
5,548,722 A * 8/1996 Jalalian
5,581,686 A * 12/1996 Koppolu et al.
5,584,035 A * 12/1996 Duggan et al.
5,634,129 A * 5/1997 Dickinson
5,669,005 A * 9/1997 Curbow

FOREIGN PATENT DOCUMENTS

EP 0 631 456 A2 * 12/1994
GB 2 242 293 * 1/1990

OTHER PUBLICATIONS

Reinhardt, Andy, "The Network with Smarts" BYTE, Oct. 1994, pp. 51–64.*
Lippman, Stanley B., "C++ Primer" 2nd edition, Addison–Wesley, 1991, pp. 394–397.*
Potel et al; The Architecture of the Taligent System; Dr. Dobbs Journal on CD–ROM, SP 94.*
Rush, Jeff; OpenDoc; Dr. Dobb's Journal on CD–ROM, SP 94.*
Piersol, Kurt; A Close–Up of OpenDoc; AIXpert, Jun. 1994.*

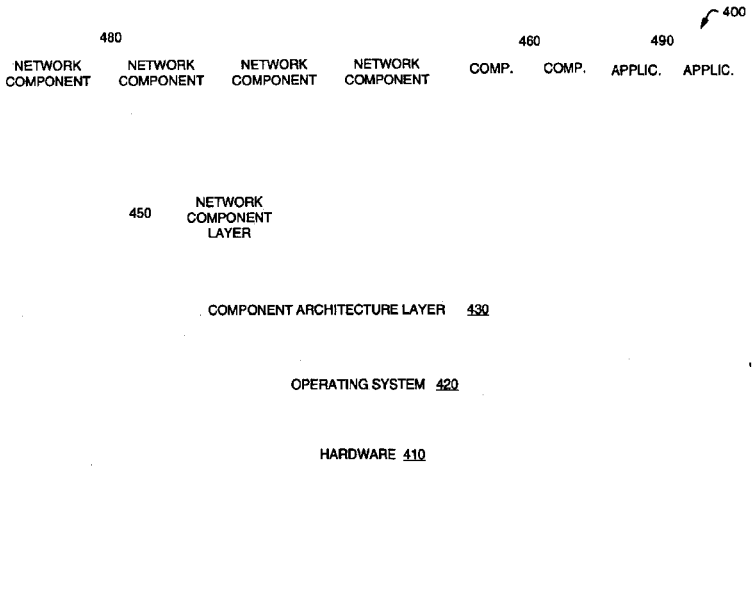
(Continued)

*Primary Examiner*—William Thomson
(74) *Attorney, Agent, or Firm*—Fenwick & West LLP

(57) **ABSTRACT**

An extensible and replaceable network-oriented component system provides a platform for developing networking navigation components that operate on a variety of hardware and software computer systems. These navigation components include key integrating components along with components configured to deliver conventional services directed to computer networks, such as Gopher-specific and Web-specific components. Communication among these components is achieved through novel application programming interfaces (APIs) to facilitate integration with an underlying software component architecture. Such a high-modular cooperating layered-arrangement between the network component system and the component architecture allows any existing component to be replaced, and allows new components to be added, without affecting operation of the network component system.

**20 Claims, 8 Drawing Sheets**



OTHER PUBLICATIONS

Schmidt et al; "An object–oriented framework for developing network server daemons", C+++ World Conference, pp. 1–15, Oct. 1993.*

"Leveraging object–oriented frameworks", Taligent white paper, 1993.*

Andert, Glerk; "Object–Frameworks in the Taligent OS", IEEE electronic Library, pp. 112–121, 1994.*

Helm et al, "Integrating information retrieval and domain specific approaches for browsing and retrieval in object–oriented class libraries", ACM Digital Library, 1991.*

Monnard et al; An object–oriented scripting environment for the WEBSs electronic book system' ACM Digital Library, 1992.*

Norr, Henry. "Cyberdog could be a breakthrough if it's Kept on a leash", MacWeek, Nov. 14, 1994, v8, n45, p. 50.*

Hess, Robert, "Cyberdog to fetch Internet Resources for Open Doc apps." MacWeek, Nov. 7, 1994, v8, n44, p. 44.*

Harkey et al, "Object component suites", Datamation, Feb. 15, 1995, v41, n3, p. 44.*

Prosise, Jeff, "Much ado about object", PC Magazine, Feb. 7, 1995, v14, n3, p. 257.*

Bonner, Paul, "Component software: putting the pieces together", Computer Shopper, Sep. 1994, v14, n9, p. 532.*

Gruman, Galen, "OpenDoc & OLE 2.0", MacWorld, Nov. '94, v11, n11, p. 96.*

Spiegel, Leo "OLE promises barrier–free computing", InfoWorld, Mar. 6, '95, v17, n10, p. 53.*

Develop, The Apple Technical Journal, "Building an OpenDoc Part Handler", Issue 19, Sep. 1994, pp. 6–16.*

S.H. Goldberg and J.A. Mounton, Jr. A Base for Portable Communications Software, IBM Systems Journal, vol. 30 (1991) No. 3, Armonk, NY, pp. 259–279.*

E.C. Arnold and D.W. Brown, Object Oriented Software Technologies Applied to Switching System Architecture and Software Development Processes, AT&T Bell Laboratories, Naperville, IL, vol. II, pp. 97–106.*

* cited by examiner

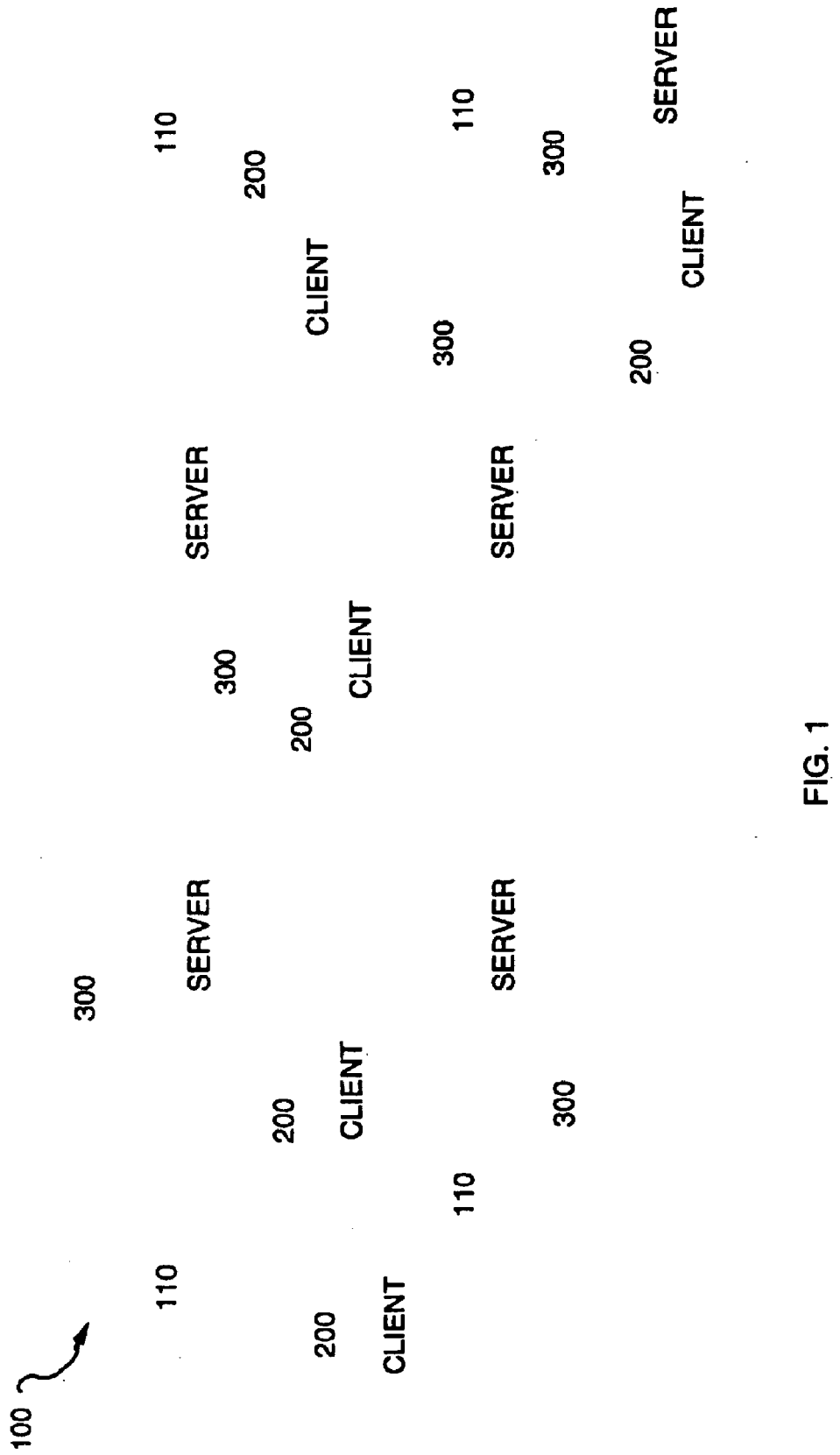

100
110
200
CLIENT
300
SERVER
110
200
CLIENT
300
SERVER
110
200
CLIENT
300
SERVER
110
200
CLIENT
300
SERVER

FIG. 1

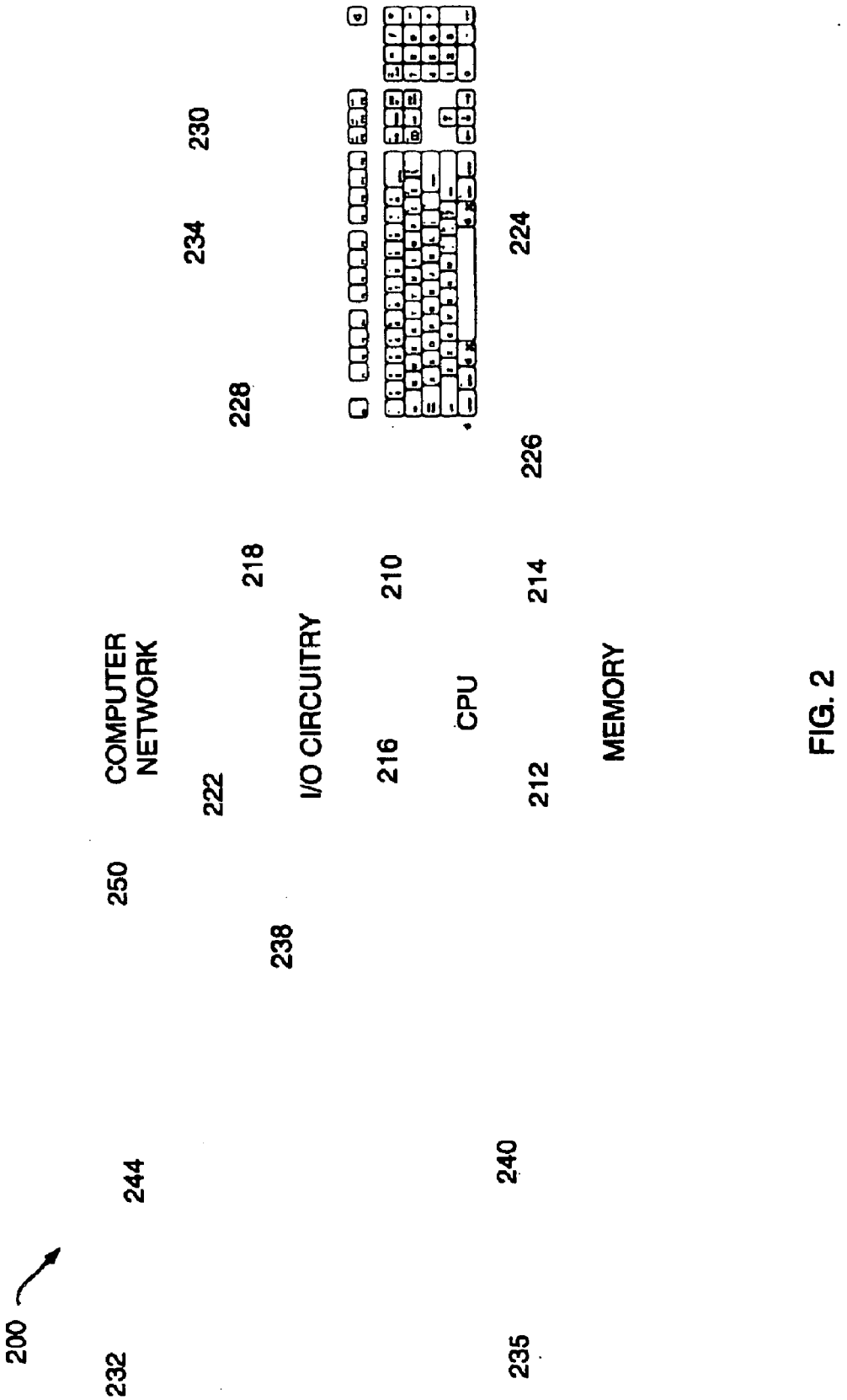

200
232
COMPUTER NETWORK
250
244
222
I/O CIRCUITRY
238
218
216
CPU
210
212
MEMORY
214
240
235
228
234
230
224
226

FIG. 2

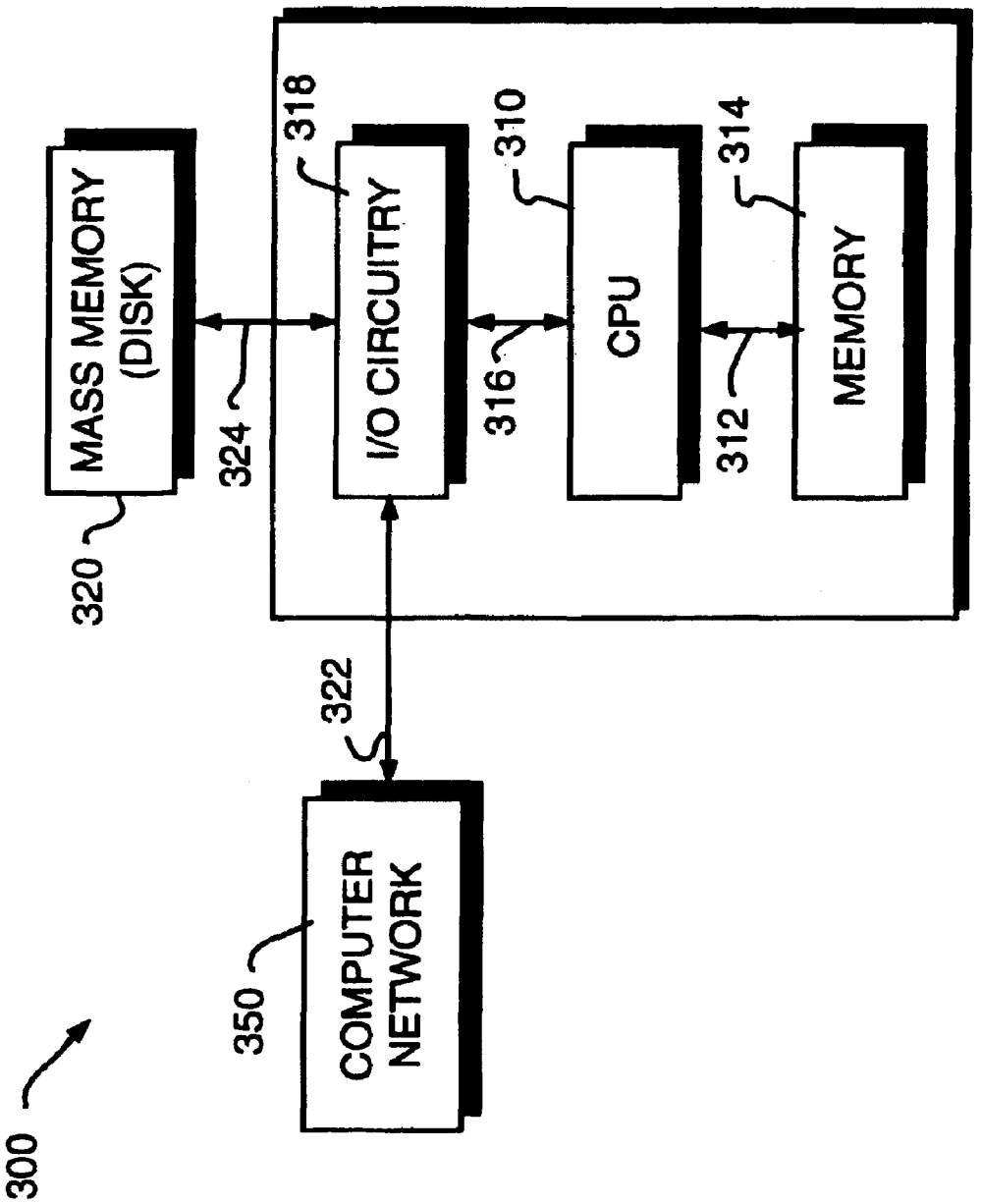

MASS MEMORY (DISK)
320
324
I/O CIRCUITRY
318
316
CPU
310
312
MEMORY
314
322
COMPUTER NETWORK
350
300

FIG. 3

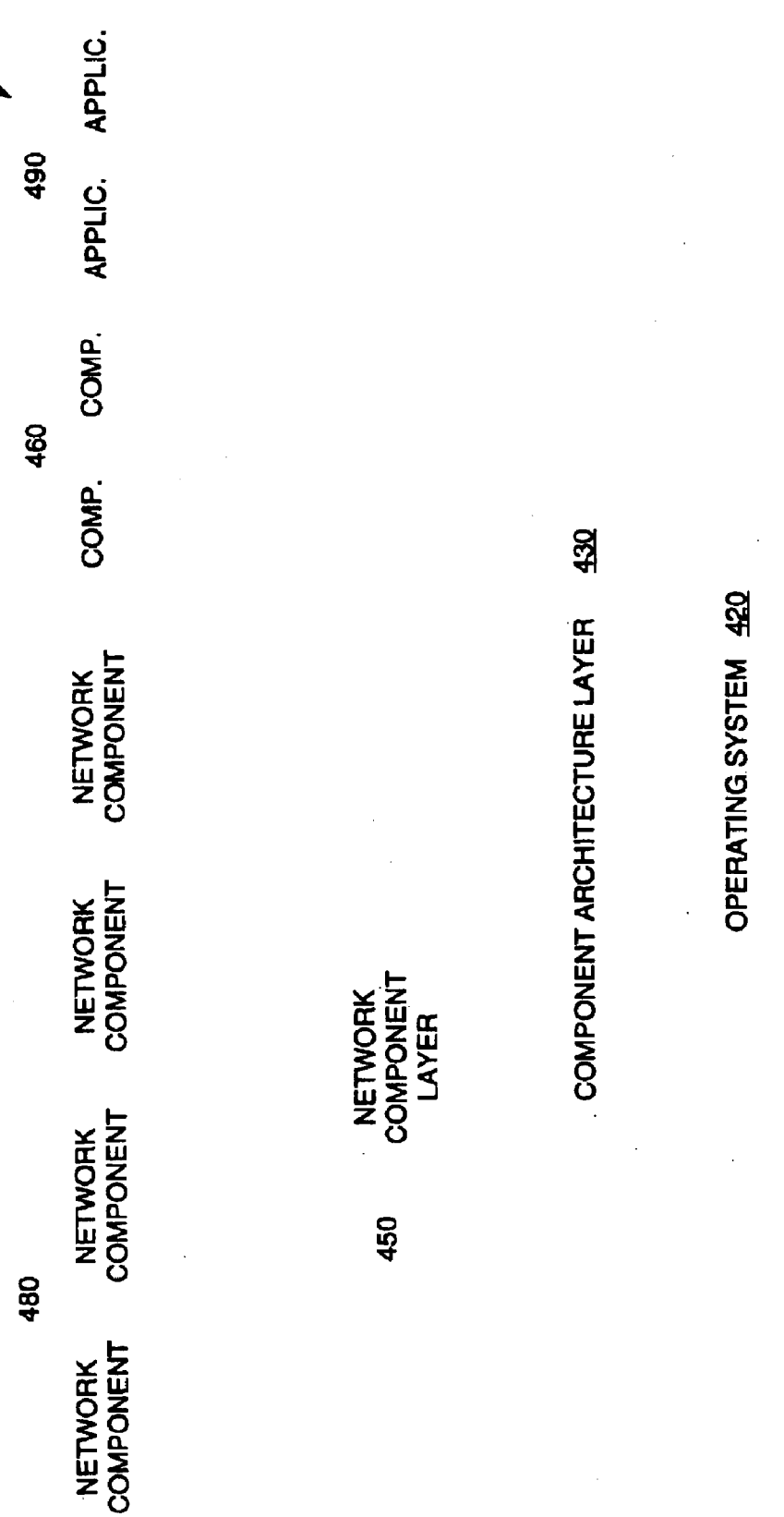

400
490
APPLIC.
APPLIC.
460
COMP.
COMP.
NETWORK COMPONENT
NETWORK COMPONENT
480
NETWORK COMPONENT
NETWORK COMPONENT
450
NETWORK COMPONENT LAYER
COMPONENT ARCHITECTURE LAYER 430
OPERATING SYSTEM 420
HARDWARE 410

FIG. 4

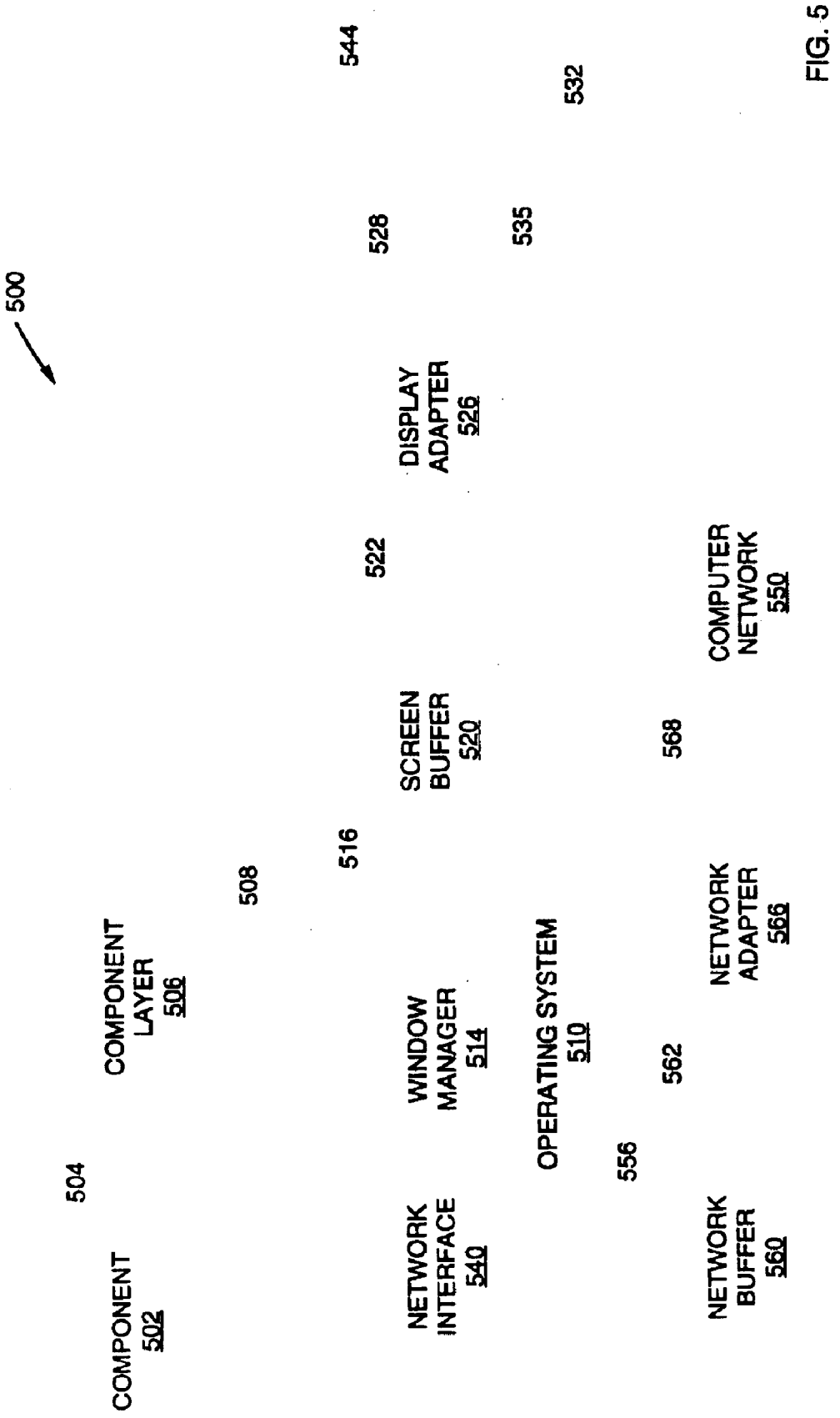

500
504
COMPONENT 502
COMPONENT LAYER 506
508
516
NETWORK INTERFACE 540
WINDOW MANAGER 514
OPERATING SYSTEM 510
556
562
NETWORK BUFFER 560
NETWORK ADAPTER 566
568
SCREEN BUFFER 520
522
DISPLAY ADAPTER 526
528
535
532
544
COMPUTER NETWORK 550

FIG. 5

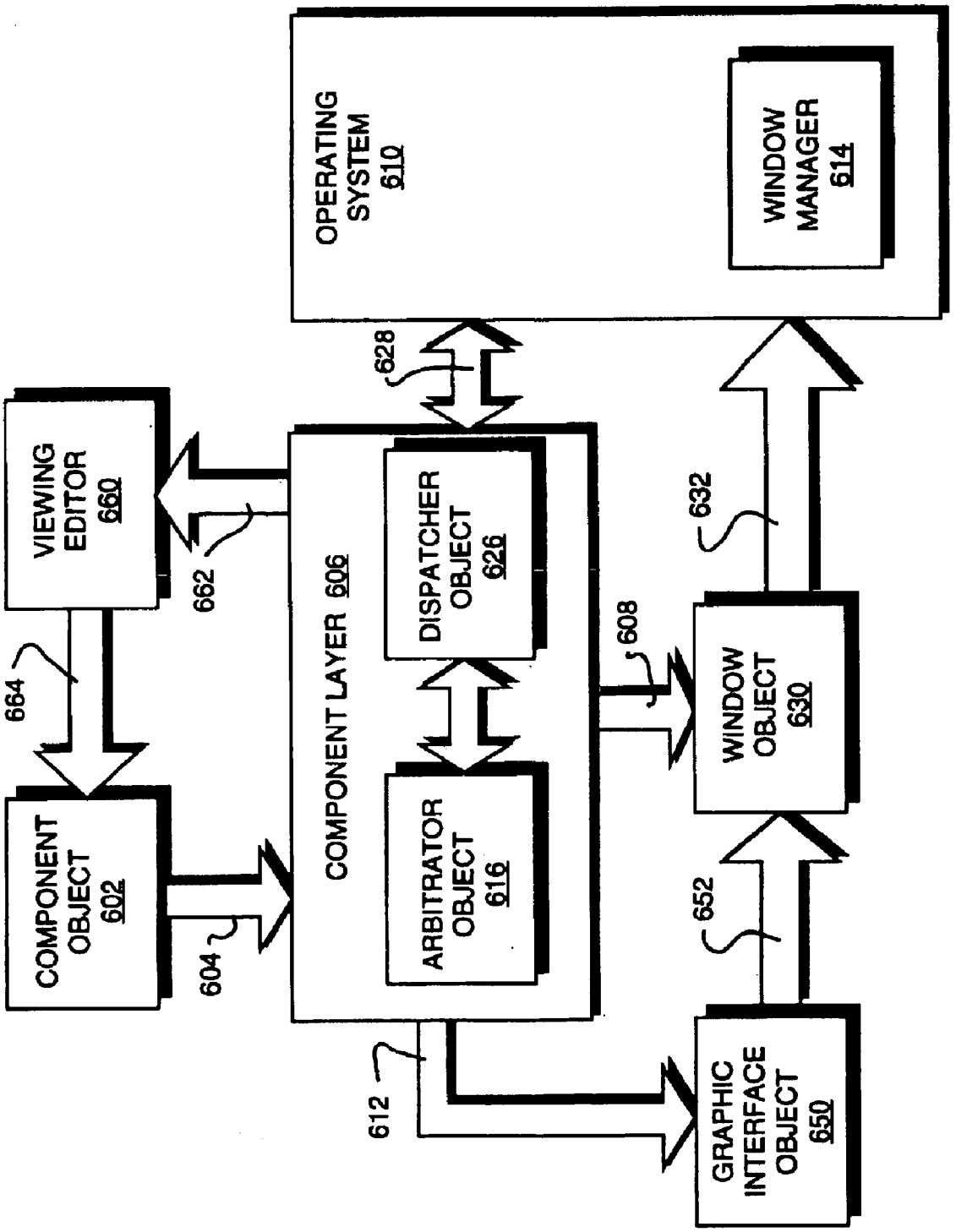

OPERATING SYSTEM 610
WINDOW MANAGER 614
628
VIEWING EDITOR 660
662
COMPONENT LAYER 606
DISPATCHER OBJECT 626
632
608
WINDOW OBJECT 630
664
COMPONENT OBJECT 602
604
ARBITRATOR OBJECT 616
652
612
GRAPHIC INTERFACE OBJECT 650

FIG. 6

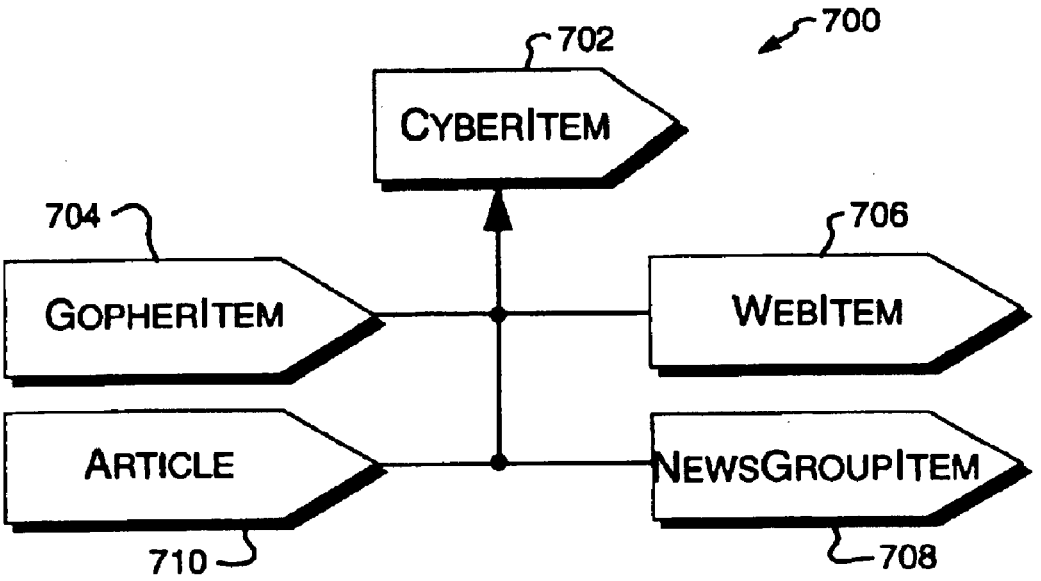

700
702
CYBERITEM
704
GOPHERITEM
706
WEBITEM
ARTICLE
NEWSGROUPITEM
710
708

FIG. 7

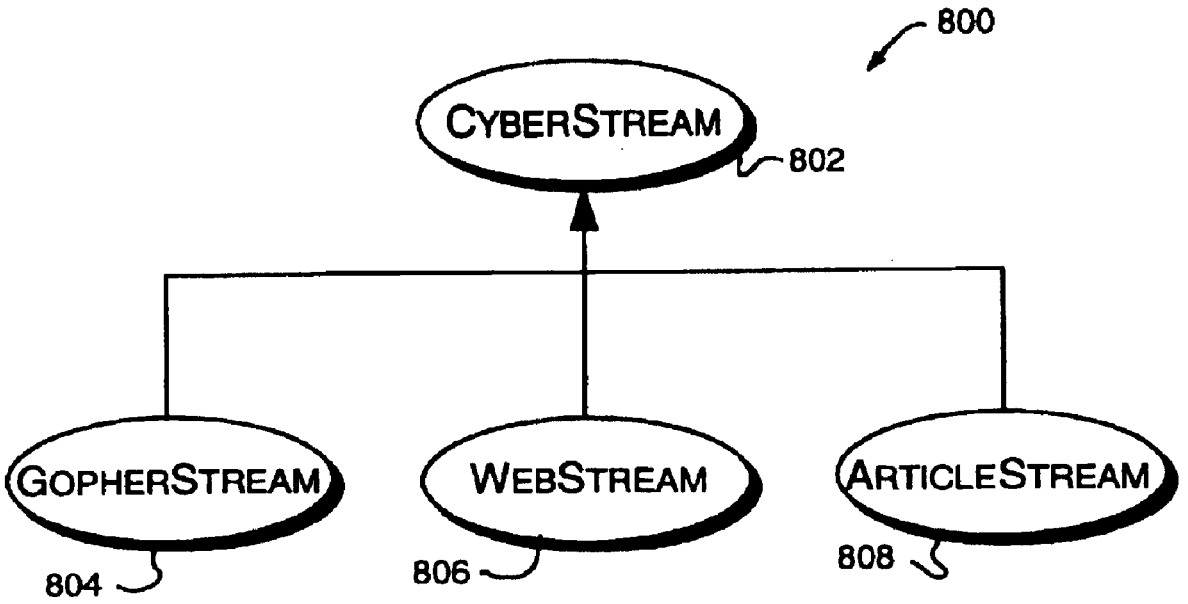

800
CYBERSTREAM
802
GOPHERSTREAM
WEBSTREAM
ARTICLESTREAM
804
806
808

FIG. 8

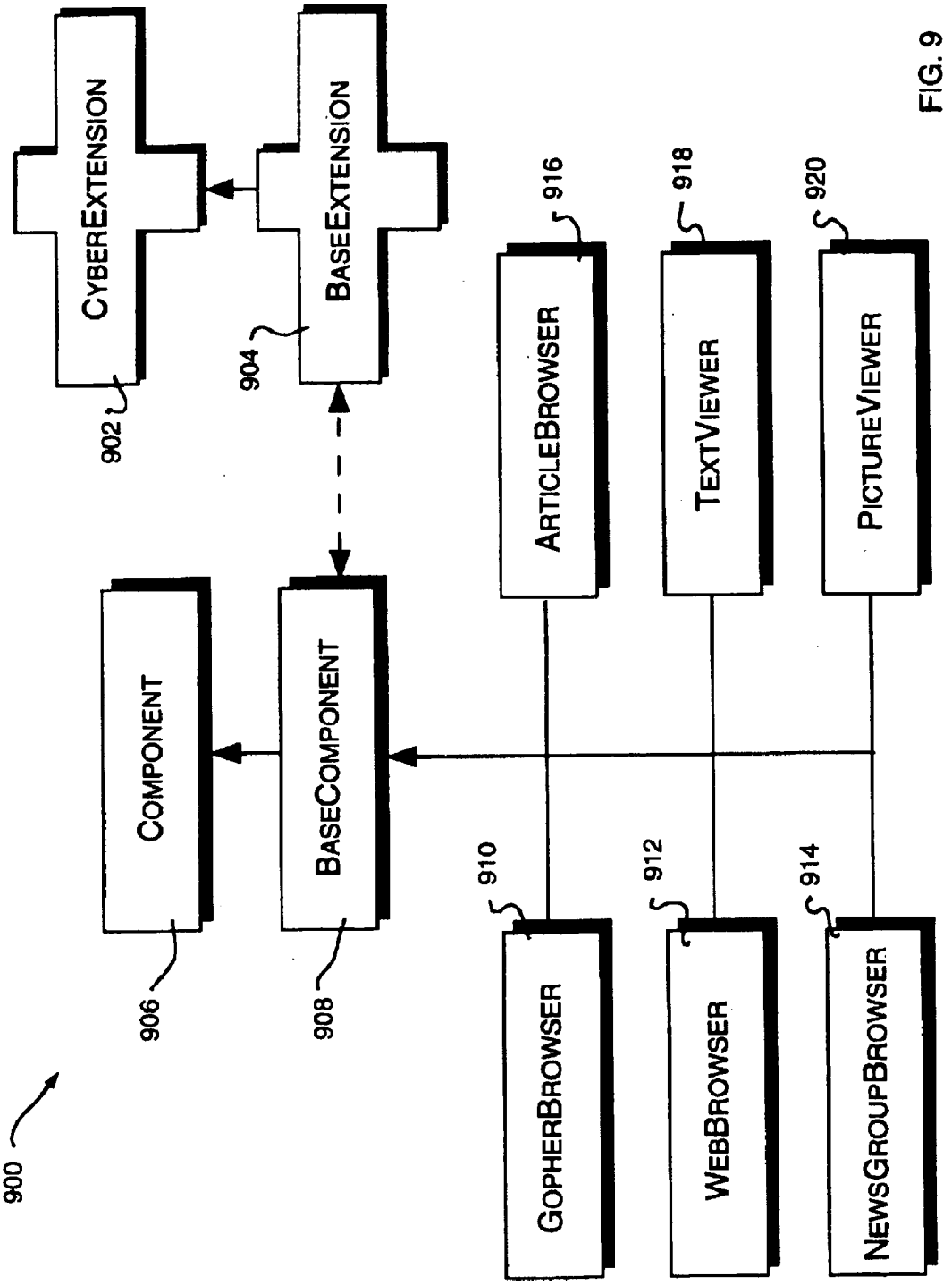

CyberExtension
BaseExtension
902
904
ArticleBrowser
TextViewer
PictureViewer
916
918
920
Component
BaseComponent
906
908
GopherBrowser
WebBrowser
NewsGroupBrowser
910
912
914
900

FIG. 9

# EXTENSIBLE, REPLACEABLE NETWORK COMPONENT SYSTEM

**Matter enclosed in heavy brackets [ ] appears in the original patent but forms no part of this reissue specification; matter printed in italics indicates the additions made by reissue.**

## CROSS REFERENCE TO RELATED APPLICATIONS

This invention is related to the following copending U.S. Patent Applications:

U.S. patent application Ser. No. 08/435,374, titled REPLACEABLE AND EXTENSIBLE NOTEBOOK COMPONENT OF A NETWORK COMPONENT SYSTEM.

U.S. patent application Ser. No. 08/435,862, titled REPLACEABLE AND EXTENSIBLE LOG COMPONENT OF A NETWORK COMPONENT SYSTEM;

U.S. patent application Ser. No. 08/435,213, titled REPLACEABLE AND EXTENSIBLE CONNECTION DIALOG COMPONENT OF A NETWORK COMPONENT SYSTEM;

U.S. patent application Ser. No. 08/435,671, titled EMBEDDING INTERNET BROWSER/BUTTONS WITHIN COMPONENTS OF A NETWORK COMPONENT SYSTEM; and

U. S. patent application Ser. No. 08/435,880, tilted ENCAPSULATED NETWORK ENTITY REFERENCE OF A NETWORK COMPONENT SYSTEM, each of which was filed on May 5, 1995 and assigned to the assignee of the present invention.

## FIELD OF THE INVENTION

This invention relates generally to computer networks and, more particularly, to an architecture for building Internet-specific services.

## BACKGROUND OF THE INVENTION

The Internet is a system of geographically distributed computer networks interconnected by computers executing networking protocols that allow users to interact and share information over the networks. Because of such wide-spread information sharing, the Internet has generally evolved into an "open" system for which developers can design software for performing specialized operations, or services, essentially without restriction. These services are typically implemented in accordance with a client/server architecture, wherein the clients, e.g., personal computers or workstations, are responsible for interacting with the users and the servers are computers configured to perform the services as directed by the clients.

Not surprisingly, each of the services available over the Internet is generally defined by its own networking protocol. A protocol is a set of rules governing the format and meaning of messages or "packets" exchanged over the networks. By implementing services in accordance with the protocols, computers cooperate to perform various operations, or similar operations in various ways, for users wishing to "interact" with the networks. The services typically range from browsing or searching the information-having a particular data format using a particular protocol to actually acquiring information of a different format in accordance with a different protocol.

For example, the file transfer protocol (FTP) service facilitates the transfer and sharing of files across the Internet.



The Telnet service allows users to log onto computers coupled to the networks, while the network protocol provides a bulletin-board service to its subscribers. Furthermore, the various data formats of the information available on the Internet include JPEG images, MPEG movies and μ-law sound files.

Coincided with the design of these services has been the development of applications for implementing the services on the client/server architecture. Accordingly, applications are available for users to obtain files from computers connected to the Internet using the FTP protocol. Similarly, individual applications allow users to log into remote computers (as though they were logging in from terminals attached to those computers) using the Telnet protocol and, further, to view JPEG images and MPEG movies. As a result, there exists a proliferation of applications directed to user activity on the Internet.

A problem with this vast collection of application-specific protocols is that these applications are generally unorganized, thus requiring users to plod through them in order to satisfyingly, and profitably, utilize the Internet. Such lack of uniformity is time consuming and disorienting to users that want to access particular types of information but are forced to use unfamiliar applications. Because of the enormous amount of different types of information available on the Internet and the variety of applications needed to access those information types, the experience of using the Internet may be burdensome to these users.

An alternative to the assortment of open applications for accessing information on the Internet is a "closed" application system, such as Prodigy, CompuServe or America Online. Each of these systems provide a fill range of well-organized services to their subscribers; however, they also impose restrictions on the services developers can offer for their systems. Such constraint of "new" service development may be an unreasonable alternative for many users.

Two fashionable services for accessing information over the Internet are Gopher and the World-Wide Web ("Web"). Gopher consists of a series of Internet servers that provide a "list-oriented" interface to information available on the networks, the information is displayed as menu items in a hierarchical manner. Included in the hierarchy of menus are documents, which can be displayed or saved, and searchable indexes, which allow users to type keywords and perform searches.

Some of the menu items displayed by Gopher are links to information available on other servers located on the networks. In this case, the user is presented with a list of available information documents that can be opened. The opened documents may display additional lists or they may contain various data-types, such as pictures or text, occasionally, the opened documents may "transport" the user to another computer on the Internet.

The other popular information services on the Internet is the Web. Instead of providing a user with a hierarchical list-oriented view of information, the Web provides the user with a "linked-hypertext" view. Metaphorically, the Web perceives the Internet as a vast book of pages, each of which may contain pictures, text, sound, movies or various other types of data in the form of documents. Web documents are written in HyperText Markup Language (HTML) and Web servers transfer HTML documents to each other through the HyperText Transfer Protocol (HTTP).

The Web service is essentially a means for naming sources of information on the Internet. Armed with such a general naming convention that spans the entire network

system, developers are able to build information servers that potentially any user can access. Accordingly, Gopher servers, HTTP servers, FTP servers, and E-mail servers have been developed for the Web. Moreover, the naming convention enables users to identify resources (such as directories and documents) on any of these servers connected to the Internet and allow access to those resources.

As an example, a user "traverses" the Web by following hot items of a page displayed on a graphical Web browser. These hot items are hypertext links whose presence are indicated on the page by visual cues, e.g., underlined words, icons or buttons. When a user follows a link (usually by clicking on the cue with a mouse), the browser displays the target pointed to by the link which, in some cases, may be another HTML document.

The Gopher and Web information services represent entirely different approaches to interacting with information on the Internet. One follows a list-approach to information that "looks" like a telephone directory service, while the other assumes a page-approach analogous to a tabloid newspaper. However, both of these approaches include applications for enabling users to browse information available on Internet servers. Additionally, each of these applications has a unique way of viewing and accessing the information on the servers.

Netscape Navigator™ ("Netscape")is an example of a monolithic Web browser application that is configured to interact with many of the previously-described protocols, including HTFF, Gopher and FTP. When instructed to invoke an application that uses one of these protocols, Netscape "translates" the protocol to hypertext. This translation places the user farther away from the protocol designed to run the application and, in some cases, actually thwarts the user's Internet experience. For example, a discussion system requiring an interactive exchange between participants may be bogged down by hypertext translations.

The Gopher and Web services may further require additional applications to perform specific functions, such as playing sound or viewing movies, with respect to the data types contained in the documents. For example, Netscape employs helper applications for executing applications having data formats it does not "understand". Execution of these functions on a computer requires interruption of processing and context switching (i.e., saving of state) prior to invoking the appropriate applications. Thus, if a user operating within the Netscape application "opens" an MPEG movie, that browsing application number must be saved (e.g., to disk) prior to opening an appropriate MPEG application, e.g., Sparkle, to view the image. Such an arrangement is inefficient and rather disruptive to processing operations of the computer.

Typically, a computer includes an operating system and application software which, collectively, control the operations of the computer. The applications are preferably task-specific and independent, e.g., a word processor application edits text, a drawing application edits drawings and a database application interacts with information stored on a database storage unit. Although a user can move data from one application to the other, such as by copying a drawing into a word processing file, the independent applications must be invoked to thereafter manipulate that data.

Generally, the application program presents information to a user through a window of a graphical user interface by drawing images, graphics or text within the window region. The user, in turn, communicates with the application by "pointing" at graphical objects in the window with a pointer that is controlled by a hand-operated pointing device, such as a mouse, or by pressing keys of a keyboard.

The graphical objects typically included with each window region are sizing boxes, buttons and scroll bars. These objects represent user interface elements that the user can point at with the pointer (or a cursor) to select or manipulate. For example, the user may manipulate these elements to move the windows around on the display screen, and change their sizes and appearances so as to arrange the window in a convenient manner. When the elements are selected or manipulated, the underlying application program is informed, via the window environment, that control has been appropriated by the user.

A menu bar is a further example of a user interface element that provides a list of menus available to a user. Each menu, in turn, provides a list of command options that can be selected merely by pointing to them with the mouse-controlled pointer. That is, the commands may be issued by actuating the mouse to move the pointer onto or near the command selection, and pressing and quickly releasing, i.e., "clicking" a button on the mouse.

In contrast to this typical application-based computing environment, a software component architecture provides a modular document-based computing arrangement using tools such as viewing editors. The key to document-based computing is the compound document, i.e., a document composed of many different types of data sharing the same file. The types of data contained in a compound document may range from text, tables and graphics to video and sound. Several editors, each designed to handle a particular data type of format, can work on the contents of the document at the same time, unlike the application-based computing environment.

Since many editors may work together on the same document, the compound document is apportioned into individual modules of context for manipulation by the editors. The compound-nature of the document is realized by embedding these modules within each other to create a document having a mixture of data types. The software component architecture provides the foundation for assembling documents of differing contents and the present invention is directed to a system for extending this capability to network-oriented services.

Therefore, it is among the objects of the present invention to simplify a user's experience on computer networks without sacrificing the flexibility afforded the user by employing existing protocols and data types available on those networks.

Another object of the invention is to provide a system for users to search and access information on the Internet without extensive understanding or knowledge of the underlying protocols and data formats needed to access that information.

Still another object of the invention is to provide a document-based computing system that enables users to develop modules for services directed to information available on computer networks.

Still yet another object of the invention is to provide a platform that allows third-party developers to extend a layered network component system by building new components that seamlessly interact with the system components.

## SUMMARY OF THE INVENTION

Briefly, the invention comprises an extensible and replaceable network-oriented component system that pro-

vides a platform for developing network navigation components that operate on a variety of hardware and software computer system. These navigation components include key integrating components along with components, such as Gopher-specific and Web-specific components, configured to deliver conventional services directed to computer networks. Communication among these components is achieved through novel application programming interfaces (APIs) to facilitate integration with an underlying software component architecture. Such a highly-modular cooperating layered-arrangement between the network component system and the component architecture allows any existing component to be replaced, and allows new components to be added, without affecting operation of the novel network component system.

According to one aspect of the present invention, the novel system provides a network navigating service for browsing and accessing information available on the computer networks. The information may include various data types available from a variety of sources coupled to the computer networks. Upon accessing the desired information, component viewing editors are provided to modify or display, either visually or acoustically, the contents of the data types regardless of the source of the underlying data. Additional components and component viewing editors may be created in connection with the underlying software component architecture to allow integration of different data types and protocols needed to interact with information on the Internet.

In accordance with another aspect of the invention, the component system is preferably embodied as a customized framework having a set of interconnected abstract classes for defining network-oriented objects. These abstract classes include CyberItem, CyberStream and CyberExtension, and the objects they define are used to build the novel navigation components. Interactions among these latter components and existing components of the underlying software architecture provide the basis for the extensibility and replaceability features of the network component system.

Specifically, CyberItem is an object abstraction which represents a "resource on a computer-network", but which may be further expanded to include resources available at any accessible location. CyberStream is an object abstraction representing a method for downloading information from a remote location on the computer network, while CyberExtension represents additional behaviors provided to the existing components for integration with the network component system.

The novel network system captures the essence of a "component-based" approach to browsing and retrieving network-oriented information as opposed to the monolithic application-based approach of prior browsing systems. Such a component-based system has a number of advantages. First, if a user does not like the way a particular component operates, that component can be replaced with a different component provided by another developer. In contrast, if a user does not like the way a monolithic application handles certain protocols, the only resource is to use another service because the user cannot modify the application to perform the protocol function in a different manner. Clearly, the replaceability feature of the novel network component system provides a flexible alternative to the user.

Second, the use of components is substantially less disruptive than using helper applications in situations where a monolithic application confronts differing data types and formats. Instead of "switching" application layers, the novel network system merely invokes the appropriate component and component viewing editor configured to operate with the data type and format. Such "seamless" integration among components is a significant feature of the modular cooperating architecture described herein.

A third advantage of the novel network system is directed to the cooperating relationship between the system and the underlying software computer architecture. Specifically, the novel network components are based on the component architecture technology to therefore ensure cooperation between all components in an integrated manner. The software component architecture is configured to operate on a plurality of computers, and is preferably implemented as a software layer adjoining the operating system.

## BRIEF DESCRIPTION OF THE DRAWINGS

The above and further advantages of the invention may be better understood by referring to the following description in conjunction with the accompanying drawings in which:

FIG. **1** is a block diagram of a network system including a collection of computer networks interconnected by client and server computers;

FIG. **2** is a block diagram of a client component, such as a personal computer, on which the invention may advantageously operate;

FIG. **3** is a block diagram of a server computer of FIG. **1**;

FIG. **4** is a highly schematized block diagram of a layered component computing arrangement in accordance with the invention;

FIG. **5** is a schematic illustration of the interaction of a component, a software component layer and an operating system of the computer of FIG. **2**;

FIG. **6** is a schematic illustration of the interaction between a component, a component layer and a window manager in accordance with the invention;

FIG. **7** is a simplified class hierarchy diagram illustrating a base class CyberItem, and its associated subclasses, used to construct network component objects in accordance with the invention;

FIG. **8** is a simplified class hierarchy diagram illustrating a base class CyberStream, and its associated subclasses, in accordance with the invention; and

FIG. **9** is a simplified class hierarchy diagram illustrating a base class CyberExtension, and its associated subclasses, in accordance with the present invention.

## DETAILED DESCRIPTION OF ILLUSTRATIVE EMBODIMENT

FIG. **1** is a block diagram of a network system **100** comprising a collection of computer networks **110** interconnected by client computers ("clients") **200**, e.g., workstations or personal computers, and server computers ("servers") **300**. The servers are typically computers having hardware and software elements that provide resources or services for use by the clients **200** to increase the efficiency of their operations. It will be understood to those skilled in the art that, in an alternate embodiment, the client and server may exist on the same computer; however, for the illustrative embodiment described herein, the client and server are separate computers.

Several types of computer networks **110**, including local area networks (LANs) and wide area networks (WANs), may be employed in the system **100**. A LAN is a limited area network that typically consists of a transmission medium,

such as coaxial cable or twisted pair, while a WAN may be a public or private telecommunications facility that interconnects computers widely dispersed. In the illustrative embodiment, the network system **100** is the Internet system of geographically distributed computer networks.

Computers coupled to the Internet typically communicate by exchanging discrete packets of information according to predefined networking protocols. Execution of these networking protocols allow users to interact and share information across the networks. As an illustration, in response to a user's request for a particular service, the client **200** sends an appropriate information packet to the server **300**, which performs the service and returns a result back to the client **200**.

FIG. **2** illustrates a typical hardware configuration of a client **200** comprising a central processing unit (CPU) **210** coupled between a memory **214** and input/output (I/O) circuitry **218** by bidirectional buses **212** and **216**. The memory **214** typically comprises random access memory (RAM) for temporary storage of information and read only memory (ROM) for permanent storage of the computer's configuration and basic operating commands, such as portions of an operating system (not shown). As described further herein, the operating system controls the operations of the CPU **210** and client computer **200**.

The I/O circuitry **218**, in turn, connects the computer to computer networks, such as the Internet computer networks **250**, via a bidirectional bus **222** and to cursor/pointer control devices, such as keyboard **224** (via cable **226**) and a mouse **230** (via cable **228**). The mouse **230** typically contains at least one button **234** operated by a user of the computer. A conventional display monitor **232** having a display screen **235** is also connected to I/O circuitry **218** via cable **238**. A pointer (cursor) **240** is displayed on windows **244** of the screen **235** and its position is controllable via the mouse **230** or the keyboard **224**, as is well-known. Typically, the I/O circuitry **218** receives information, such as control and data signals, from the mouse **230** and keyboard **224**, and provides that information to the CPU **210** for display on the screen **235** or, as described further herein, for transfer over the Internet **250**.

FIG. **3** illustrates a typical hardware configuration of a server **300** of the network system **100**. The server **300** has many of the same units as employed in the client **200**, including a CPU **310**, a memory **314**, and I/O circuitry **318**, each of which are interconnected by bidirectional buses **312** and **316**. Also, the I/O circuitry connects the computer to computer networks **350** via a bidirectional bus **322**. These units are configured to perform functions similar to those provided by their corresponding units in the computer **200**. In addition, the server typically includes a mass storage unit **320**, such as a disk drive, connected to the I/O circuitry **318** via bidirectional bus **324**.

It is to be understood that the I/O circuits within the computers **200** and **300** contain the necessary hardware, e.g., buffers and adapters, needed to interface with the control devices, the display monitor, the mass storage unit and the networks. Moreover, the operating system includes the necessary software drivers to control, e.g., network adapters within the I/O circuits when performing I/O operations, such as the transfer of data packets between the client **200** and server **300**.

The computers are preferably personal computers of the Macintosh® series of computers sold by Apple Computer Inc., although the invention may also be practiced in the context of other types of computers, including the IBM®G) series of computers sold by International Business Machines Corp. These computers have resident thereon, and are controlled and coordinated by, operating system software, such as the Apple® System 7®, IBM OS2®, or the Microsoft® Windows® operating systems.

As noted, the present invention is based on a modular document computing arrangement as provided by an underlying software components architecture, rather than the typical application-based environment of prior computing systems. FIG. **4** is a highly schematized diagram of the hardware and software elements of a layered component computing arrangement **400** that includes the novel network-oriented component system of the invention. At the lowest level there is the computer hardware, shown as layer **410**. Interfacing with the hardware is a conventional operating system layer **420** that includes a window manager, a graphic system, a file system and network-specific interfacing, such as a TCP/IP protocol stack and an Appletalk protocol stack.

The software component architecture is preferably implemented as a component architecture layer **430**. Although it is shown as overlaying the operating system **420**, the component architecture layer **430** is actually independent of the operating system and, more precisely, resides side-by-side with the operating system. This relationship allows the component architecture to exist on multiple platforms that employ different operating systems.

In accordance with the present invention, a novel network:oriented component layer **450** contains the underlying technology for implementing the extensible and replaceable network component system that delivers services and facilitates development of navigation components directed to computer networks, such as the Internet. As described further herein, this technology includes novel application programming interfaces (APIs) that facilitate communication among components to ensure integration with the underlying component architecture layer **430**. These novel APIs are preferably delivered in the form of objects in a class hierarchy.

It should be noted that the network component layer **450** may operate with any existing system-wide component architecture, such as the Object Linking and Embedding (OLE) architecture developed by the Microsoft Corporation; however, in the illustrative embodiment, the component architecture is preferably OpenDoc, the vendor-neutral, open standard for compound documents developed by, among others, Apple Computer, Inc.

Using tools such as viewing editors, the component architecture layer **430** creates a compound document composed of data having different types and formats. Each differing data type and format is contained in a fundamental unit called a computing part or, more generally, a "component" **460** comprised of a viewing editor along with the data content. An example of the computing component **460** may include a MacDraw component. The editor, on the other hand, is analogous to an application program in a conventional computer. That is, the editor is a software component which provides the necessary functionality to display a component's contents and, where appropriate, present a user interface for modifying those contents. Additionally, the editor may include menus, controls and other user interface elements.

According to the invention, the network component layer **450** extends the functionality of the underlying component architecture layer **430** by defining network-oriented components **480**. Included among these components are key inte-

grating components (such as notebook, log and connection dialog components) along with components configured to deliver conventional services directed to computer networks, such as Gopher-specific and Web-specific components. Moreover, the components may include FTP-specific components for transferring files across the networks. Telnet-specific components for remotely logging onto other computers, and JPEG-specific and MPEG-specific components for viewing image and movie data types and formats.

A feature of the invention is the ability to easily extend, or replace, any of the components of the layered computing arrangement **400** with a different component to provide a user with customized network-related services. As described herein, this feature is made possible by the cooperating relationship between the network component layer **450** and its underlying component architecture layer **430**. The integrating components communicate and interact with these various components of the system in a "seamlessly integrated" manner to provide basic tools for navigating the Internet computer networks.

FIG. **4** also illustrates the relationship of applications **490** to the elements of the layered computing arrangement **400**. Although they reside in the same "user space" as the components **460** and network components **480**, the applications **490** do not interact with these elements and, thus, interface directly to the operating system layer **420**. Because they are designed as monolithic, autonomous modules, applications (such as previous Internet browsers) often do not even interact among themselves. In contrast, the components of the arrangement **400** are designed to work together via the common component architecture layer **430** or, in the case of the network components, via the novel network component layer **450**.

Specifically, the invention features the provision of the extensible and replaceable network-oriented component system which, when invoked, causes actions to take place that enhance the ability of a user to interact with the computer to search for, and obtain, information available over computer networks such as the Internet. The information is manifested to a user via a window environment, such as the graphical user interface provide by System 7 or Windows, that is preferably displayed on the screen **235** (FIG. **2**) as a graphical display to facilitate interactions between the user and the computer, such as the client **200**. This behavior of the system is brought about by the interaction of the network components with a series of system software routines associated with the operating system **420**. These system routines, in turn, interact with the components architecture layer **430** to create the windows and graphical user interface elements, as described further herein.

The window environment is generally part of the operating system software **420** that includes a collection of utility programs for controlling the operation of the computer **200**. The operating system, in turn, interacts with the components to provide higher levels functionality, including a direct interface with the user. A component makes use of operating system functions by issuing a series of task commands to the operating system via the network component layer **450** or, as is typically the case, through the component architecture layer **430**. The operating system **420** then performs the requested task. For example, the component may request that a software driver of the operating system initiate transfer of a data packet over the networks **250** or that the operating system display certain information on a window for presentation to the user.

FIG. **5** is a schematic illustration of the interaction of a component **502**, software component layer **506** and an operating system **510** of a computer **500**, which is similar to, and has equivalent elements of, the client computer **200** of FIG. **2**. As noted, the network component layer **450** (FIG. **4**) is integrated with the computer architecture layer **430** to provide a cooperating architecture that allows any component to be replaced or extended, and allows new components to be added, without affecting operation of the network component system, accordingly, for purposes of the present discussion, the layers **430** and **450** may be treated as a single software component layer **506**.

The component **502**, component layer **506** and operating system **510** interact to control and coordinate the operations of the computer **500** and their interaction is illustrated schematically by arrows **504** and **508**. In order to display information on a screen display **535**, the component **502** and component layer **506** cooperate to generate and send display commands to a window manager **514** of the operating system **510**. The window manager **514** stores information directly (via arrow **516**) into a screen buffer **520**.

The window manager **514** is a system software routine that is generally responsible for managing windows **544** that the user views during operation of the network component system. That is, it is generally the task of the window manager to keep track of the location and size of the window and window areas which must be drawn and redrawn in connection with the network component system of the present invention.

Under control of various hardware and software in the system, the contents of the screen buffer **520** are read out of the buffer and provided, as indicated schematically by arrow **522**, to a display adapter **526**. The display adapter contains hardware and software (sometimes in the form of firmware) which converts the information in the screen buffer **520** to a form which can be used to drive a display screen **535** of a monitor **532**. The monitor **532** is connected to display adapter **526** by cable **528**.

Similarly, in order to transfer information as a packet over the computer networks, the component **502** and component layer **506** cooperate to generate and send network commands, such as remote procedure calls, to a network-specific interface **540** of the operating system **510**. The network interface comprises system software routines, such as "stub" procedure software and protocol stacks, that are generally responsible for forming the information into a predetermined packet format according to the specific network protocol used, e.g., TCP/IP or Apple-talk protocol.

Specifically, the network interface **540** stores the packet directly (via arrow **556**) into a network buffer **560**. Under control of the hardware and software in the system, the contents of the network buffer **560** are provided, as indicated schematically by arrow **562**, to a network adapter **566**. The network adapter incorporates the software and hardware, i.e., electrical and mechanical interchange circuits and characteristics, needed to interface with the particular computer networks **550**. The adapter **566** is connected to the computer networks **550** by cable **568**.

In a preferred embodiment, the invention described herein is implemented in an object-oriented programming (OOP) language, such as C++, using System Object Model (SOM) technology and OOP techniques. The C++ and SOM languages are well-known and many articles and texts are available which describe the languages in detail. In addition, C++ and SOM compilers are commercially available from several vendors. Accordingly, for reasons of brevity, the details of the C++ and SOM languages and the operations of their compilers will not be discussed further in detail herein.

As will be understood by those skilled in the art, OOP techniques involve the definition, creation, use and destruction of "objects". These objects are software entities comprising data elements and routines, or functions, which manipulate the data elements. The data and related functions are treated by the software as an entity that can be created, used and deleted as if it were a single item. Together, the data and functions enable objects to model virtually any real-world entity in terms of its characteristics, which can be represented by the data elements, and its behavior, which can be represented by its data manipulation functions. In this way, objects can model concrete things like computers, while also modeling abstract concepts like numbers or geometrical designs.

Objects are defined by creating "classes" which are not objects themselves, but which act as templates that instruct the compiler how to construct an actual object. A class may, for example, specify the number and type of data variables and the steps involved in the functions which manipulate the data. An object is actually created in the program by means of a special function called a "constructor" which uses the corresponding class definition and additional information, such as arguments provided during object creation, to construct the object. Likewise objects are destroyed by a special function called a "destructor". Objects may be used by manipulating their data and invoking their functions.

The principle benefits of OOP techniques arise out of three basic principles encapsulation, polymorphism and inheritance. Specifically, objects can be designed to hide, or encapsulate all, or a portion of, its internal data structure and internal functions. More specifically, during program design, a program developer can define objects in which all or some of the data variables and all or some of the related functions are considered "private" or for use only by the object itself. Other data or functions can be declared "public" or available for use by other programs. Access to the private variables by other programs can be controlled by defining public functions for an object which access the object's private data. The public functions form a controlled and consistent interface between the private data and the "outside" world. Any attempt to write program code which directly accesses the private variables causes the compiler to generate an error during program compilation which error stops the compilation process and prevents the program from being run.

Polymorphism is a concept which allows objects and functions that have the same overall format, but that work with different data, to function differently in order to produce consistent results. Inheritance, on the other hand, allows program developers to easily reuse pre-existing programs and to avoid creating software from scratch. The principle of inheritance allows a software developer to declare classes (and the objects which are later created from them) as related. Specifically, classes may be designated as subclasses of other base classes. A subclass "inherits" and has access to all of the public functions of its base classes just as if these functions appeared in the subclass. Alternatively, a subclass can override some or all of its inherited functions or may modify some or all of its inherited functions merely by defining a new function with the same form (overriding or modification does not alter the function in the base class, but merely modifies the use of the function in the subclass). The creation of a new subclass which has some of the functionality (with selective modification) of another class allows software developers to easily customize existing code to meet their particular needs.

In accordance with the present invention, the component **502** and windows **544** are "objects" created by the component layer **506** and the window manager **514**, respectively, the latter of which may be an object-oriented program. Interaction between a component, component layer and a window manager is illustrative in greater detail in FIG. **6**.

In general, the component layer **606** interfaces with the window manager **614** by creating and manipulating objects. The window manger itself may be an object which is created when the operating system is started. Specifically, the component layer creates window objects **630** that create the window manager to create associated windows on the display screen. This is shown schematically by an arrow **608**. In addition, the component layer **606** creates individual graphic interface objects **650** that are stored in each window object **630**, as shown schematically by arrows **612** and **652**. Since many graphic interface objects may be created in order to display many interface elements on the display screen, the window object **630** communicates with the window manager by means of a sequence of drawing commands issued from the window object to the window manager **614**, as illustrated by arrow **632**.

As noted, the component layer **606** functions to embed components within one another to form a component document having mixed data types and formats. Many different viewing editors may work together to display, or modify, the data contents of the documents. In order to direct keystrokes and mouse events initiated by a user to the proper components and editors, the component layer **606** includes an arbitrator **616** and a dispatcher **626**.

The dispatcher is an object that communicates with the operating system **610** to identify the correct viewing editor **660**, while the arbitrator is an object that informs the dispatcher as to which editor "owns" the stream of keystrokes or mouse events. Specifically, the dispatcher **626** receives these "human-interface" events from the operating system **610** (as shown schematically by arrow **628**) and delivers them to the correct viewing editor **660** via arrow **662**. The viewing editor **660** then modifies or displays, either visually or acoustically, the contents of the data types.

Although OOP offers significant improvements over other programming concepts, software development still requires significant outlays of time and effort, especially if no pre-existing software is available for modulation. Consequently, a prior art approach has been to provide a developer with a set of preferred, interconnected classes which create a set of objects and additional miscellaneous routines that are all directed to performing commonly-encountered tasks in a particular environment. Such predefined classes and libraries are typically called "frameworks" and essentially provide a pre-fabricated structure for a working document.

For example, a framework for a user interface might provide a set of predefined graphic interface objects which create windows, scroll bars, menus, etc. and provide the support and "default" behavior for these interface objects. Since frameworks are based on object-oriented techniques, the predefined classes can be used as base classes and the built-in default behavior can be inherited by developer-defined subclasses and either modified or overridden to allow developers to extend the framework and create customized solutions in a particular area of expertise. This object-oriented approach provides a major advantage over traditional programming since the programmer is not changing the original program, but rather extending the capabilities of that original program. In addition, developers are not blindly working through layers of code because the framework providers architectural guidance and modeling and, at the same time, frees the developers to supply specific actions unique to the problem domain.

There are many kinds of framework available, depending on the level of the system involved and the kind of problem to be solved. The types of frameworks range from high-level frameworks that assist in developing a user interface, to lower-level frameworks that provide basic system software services such as communications, printing, file systems support, graphics, etc. Commercial examples of application-type frameworks include MacApp (Apple), Bedrock (Symantec), OWL (Borland), NeXT Step App Kit (NeXT) and Smalltalk-80 MVC (ParcPlace).

While the framework approach utilizes all the principles of encapsulation, polymorphism, and inheritance in the object layer, and is a substantial improvement over other programming techniques, there are difficulties which arise. These difficulties are caused by the fact that it is easy for developers to reuse their own objects, but it is difficult for the developers to use objects generated by other programs. Further, frameworks generally consist of one or more object "layers" on top of a monolithic operating system and even with the flexibility of the object layer, it is still often necessary to directly interact with the underlying system by means of awkward procedure calls.

In the same way that a framework provides the developer with prefab functionality for a document, a system framework, such as that included in the preferred embodiment, can provide a prefab functionality for system level services which developers can modify or override to create customized solutions, thereby avoiding the awkward procedural calls necessary with the prior art framework. For example, consider a customizable network interface framework which can provide the foundation for browsing and accessing information over a computer network. A software developer who needed these capabilities would ordinarily have to write specific routines to provide them. To do this with a framework, the developer only needs to supply the characteristic and behavior of the finished output, while the framework provides the actual routines which perform the tasks.

A preferred embodiment takes the concept of frameworks and applies it throughout the entire system, including the document, component, component layer and the operating system. For the commercial or corporate developer, systems integrator, or OEM, this means all of the advantages that have been illustrated for a framework, such as MacApp, can be leveraged not only at the application level for things such as text and graphical user interfaces, but also at the system level for such services as printing, graphics, multi-media, file systems and, as described herein, network-specific operations.

Referring again to FIG. **6**, the window object **630** and the graphic interface object **650** are elements of a graphical user interface of a network component system having a customizable framework for greater enhancing the ability of a user to navigate or browse through information stored on servers coupled to the network. Moreover, the novel network system provides a platform for developing network navigation components for operation on a variety of hardware and software computer systems.

As noted, the network components are preferably implemented as objects and communication among the network component objects is effected through novel application programming interfaces (APIs). These APIs are preferably delivered in the form of objects in a class hierarchy that is extensible so that developers can create new components and editors. From an implementation viewpoint, the objects can be subclassed and can inherit from base classes to build customized components allow users to see different kinds of data using different kinds of protocols, or to create components that function differently from existing components.

In accordance with the invention, the customized framework has a set of interconnected abstract classes for defining network-oriented objects used to build these customized network components. These abstract classes include CyberItem, CyberStream and CyberExtension and the objects they define are used to build the novel network components. Interactions among these latter components and existing components of the underlying software architecture provide the basis for the extensibility and replaceability features of the network component system.

In order to further understand the operations of these network component objects, it may be useful to examine their construction together with the major function routines that comprise the behavior of the objects. In examining the objects, it is also useful to examine the classes that are used to construct the objects (as previously mentioned the classes serve as templates for the construction of objects). Thus, the relation of the classes and the functions inherent in each class can be used to predict the behavior of an object once it is constructed.

FIG. **7** illustrates a simplified class hierarchy diagram **700** of the base class CyberItem **702** used to construct the network component object **602**. In general, CyberItem is an abstraction that may represent resources available at any location accessible from the client **200**. However, in accordance with the illustrative embodiment, a CyberItem is preferably a small, self-contained object that represents a resource, such as a service, available on the Internet and subclasses of the CyberItem base class are used to construct various network component objects configured to provide such services for the novel network-oriented component system.

For example, the class GopherItem **704** may be used to construct a network component object representing a "thing in Gopher space", such as a Gopher directory, while the subclass WebItem **706** is derived from CyberItem and encapsulates a network component object representing a "thing in Web space, e.g. a Web page. Similarly, the subclass NewsGroupItem **708** may be used to construct a network object representing a newsgroup and the class Article **710** is configured to encapsulate a network component object representing an article resource on an Internet server.

Since each of the classes used to construct these network component objects are subclasses of the CyberItem base class, each class inherits the functional operators and methods that are available from that base class. For example, methods associated with the CyberItem base class for returning an icon family and a name are assumed by the subclasses to allow the network components to display CyberItem objects in a consistent manner. The methods associated with the CyberItem base class include (the arguments have been omitted for simplicity):

GetRefCount ();
IncrementRefCount ();
Release ();
SetUpFromURL ();
ExternalizeContent ();
StreamToStorageUnit ();
StreamFromStorageUnit ();
Clone ();
Compare ();
GetStringProperty ();

SetDefaultName ();
GetURL ();
GetIconSuite ();
CreateCyberStream ();
Open ();
OpenInFrame ();
FindWindow ().

In some instances, a CyberItem object may need to spawn a CyberStream object in order to obtain the actual data for the object it represents. FIG. **8** illustrates a simplified class hierarchy diagram **800** of the base class CyberStream **802**. As noted, CyberStream is an abstraction that serves as an API between a component configured to display a particular data format and the method for obtaining the actual data. This allows developers to design viewing editors that can display the content of data regardless of the protocol required to obtain that data.

For example, a developer may design a picture viewing editor that uses the CyberStream API to obtain data bytes describing a picture. The actual data bytes are obtained by a subclass of CyberStream configured to construct a component object that implements a particular protocol, such as Gopher and HTTP. That is, the CyberStream object contains the software commands necessary to create a "data stream" for transferring information from one object to another. According to the invention, a GopherStream subclass **804** is derived from the CyberStream base class and encapsulates a network object that implements the Gopher protocol, while the class WebStream **806** may be used to construct a network component configured to operate in accordance with the HTTP protocol.

The methods associated with the CyberStream class, and which are contained in the objects constructed from the subclasses, include (the arguments have been omitted for simplicity):

GetStreamStatus ();
GetTotalDataSize ();
GetStreamError ();
GetStatusString ();
OpenWithCallback ();
Open ();
GetBuffer ();
ReleaseBuffer ();
Abort ().

FIG. **9** is a simplified class hierarchy diagram of the base class CyberExtension **902** which represents additional behaviors provided to components of the underlying software component architecture. Specifically, CyberExtension are the mechanisms for adding functionality to, and extending the APIs of, existing components so that they may communicate with the novel network components. As a result, the CyberExtension base class **902** operates in connection with a Component base class **906** through their respective subclasses BaseExtension **904** and BaseComponent **908**.

The CyberExtension base class provides an API for accessing other network-specific components, such as notebooks and logs, and for supporting graphical user interface elements, such as menus. CyberExtension objects are used by components that display the contents of CyterItem objects. This includes browser-like components such as a Gopher browser or Web browser, as well as JPEG-specific components which display particular types of data such as pictures. The CyberExtension objects also keep track of the CyberItem objects which these components are responsible for displaying.

In accordance with the invention, the class GopherBrowser **910** may be used to construct a Gopher-like network browsing component and the class WebBrowser **912** may be able to construct a Web-like network browsing component. Likewise, a TextViewer subclass **918** may encapsulate a network component configured to display text and a PictureViewer subclass **920** may construct a component for displaying pictures. The methods associated with the CyberExtension class include (the arguments have been omitted for simplicity):

ICyberExtension ( );
Components displaying the contents of CyberItem object

SetCyberItem ( );
GetCyberItem ( );
GetCyberItemWindow ( );
IsCyberItemSelected ( );
GetSelectedCyberItems ( );
Notebook and Log Tools

AddCyberItemToLog ( );
ShowLogWindow ( );
IsLogWindowShown ( );
AddCyberItemToNotebook ( );
AddCyberItemsToNotebook ( );
ShowNotebookWindow ( );
IsNotebookWindowShown ( );
SetLogFinger ( );
ClearLogFinger ( );
Notebook and Log Menu Handlers

InstallServicesMenu ( );
AdjustMenus ( );
DoCommand ( ).

In summary, the novel network system described herein captures, the essence of a "comprehensive-based" approach to browsing and retrieving network-oriented information as opposed to the monolithic application-based approach of prior browsing systems. Advantages of such a component-based system include the ability to easily replace and extend components because of the cooperating relationship between the novel network-oriented component system and the underlying component architecture. This relationship also facilitates "seamless" integration and cooperation between components and component viewing editors when confronted with differing data types and formats.

While there has been shown and described an illustrative embodiment for implementing an extensible and replaceable network component system, it is to be understood that various other adaptations and modifications may be made within the spirit and scope of the invention. For example, additional system software routines may be used when implementing the invention in various applications. These additional system routines include dynamic link libraries (DLL), which are program files containing collections of window environment and networking functions designed to perform specific classes of operations. These functions are invoked as needed by the software component layer to perform the desired operations. Specifically, DLLs, which are generally well-known, may be used to interact with the component layer and window manager to provide network-specific components and functions.

The foregoing description has been directed to specific embodiments of this invention. It will be apparent, however, that other variations and modifications may be made to the described embodiments, with the attainment of some or all of their advantages. Therefor, it is the object of the appended claims to cover all such variations and modifications as come within the true spirit and scope of the invention.

What is claimed is:

**1**. An extensible and replaceable layered component computing arrangement residing on a computer coupled to a computer network, the layered arrangement comprising:

a software component architecture layer interfacing with an operating system to control the operations of the computer, the software component architecture layer defining a plurality of computing components; and

a network component layer for developing network navigation components that provide services directed to the computer network, the network component layer includes application programming interfaces; and

a first class included in the application programming interfaces to construct a first network navigation object that represents different network resources available on the computer network, wherein the network component layer coupled to the software component architecture layer in integrating relation to facilitate communication among the computing and network navigation components.

**2**. The computing arrangement of claim **1** wherein the network navigation components are objects.

**3**. The computing arrangement of claim **1** wherein the application programming interfaces further comprise a second class for constructing a second network navigation object representing a data stream for transferring information among objects of the arrangement.

**4**. The computing arrangement of claim **3** wherein the first network navigation object is an Item object and the second network navigation object is a Stream object, and wherein the Item object spawns the Stream object to obtain information from the network resource that the Item object represents.

**5**. The computing arrangement of claim **3** wherein the application programming interfaces further comprise a third class for constructing a third network navigation object representing additional behaviors provided to computing components of the software component architecture layer to thereby enable communication between the computing components and the network navigation components.

**6**. An extensible and replaceable layered component computing arrangement for providing services directed to information available on computer networks, the computing arrangement comprising:

a processor;

an operating system;

a software component architecture layer coupled to the operating system to control the operations of the processor, the software component architecture layer defining a plurality of computing components; and

a network component layer for creating network navigation components configured to search and obtain information available on the computer networks, the network component layer includes application programming interfaces; and

means for constructing a network navigation component that represents different resources available on the computer network, wherein the network component layer is integrally coupled to the software component architecture layer to ensure communication among the computing and network navigation components.

**7**. The computing arrangement of claim **6** wherein the network component layer and software component architecture layer comprise means for embedding components within one another to form a compound document having mixed data types and formats.

**8**. The computing arrangement of claim **6** wherein the application programming interfaces comprise means for constructing a network navigation component that implements a protocol.

**9**. The computing arrangement of claim **6** wherein the application programming interfaces comprise means for constructing a network navigation component that provides additional functionality to existing computing components to enable communication among the components.

**10**. The computing arrangement of claim **9** wherein the computing component comprises a computing part having a viewing editor and data content.

**11**. The computing arrangement of claim **10** wherein the computing component functions to one of transfer files over the networks, remotely log onto another computer coupled to the networks and view images on a screen of the computing arrangement.

**12**. The computing arrangement of claim **10** wherein the network navigation component comprises a browsing component.

**13**. The computing arrangement of claim **10** wherein the network navigation component comprises a component for one of displaying text and displaying movies on a screen of the computing arrangement.

*14. An extensible and replaceable layered component computing arrangement residing on a computer adapted to be coupled on a computer network, the layered arrangement comprising:*

*a software component architecture layer interfacing with an operating system to control the operations of the computer, the software component architecture layer defining a plurality of computing components;*

*a network component layer adapted to be coupled to at least one network navigation component that provides a service directed to the computer network, the network component layer including an application programming interface; and*

*a number of interconnected abstract classes included in the application programming interface, at least on abstract class for defining a network navigation object that represents a resource available on the computer network, the network component layer coupled to the software component architecture layer to facilitate communication among the network navigation component and at least one computing component.*

*15. The layered arrangement of claim 14, wherein the abstract class defines a network navigation object that represents a method of downloading information from a remote location on the computer network.*

*16. The layered arrangement of claim 14, wherein the abstract class defines a network navigation object that represents additional behaviors provided to the computing components of the software component architecture layer for integrating with the network component layer.*

*17. The layered arrangement of claim 14, wherein the network navigation object is adapted to browse the computer network.*

*18. The layered arrangement of claim 14, wherein the network navigation object is adapted to display text on a computer display.*

*19. The layered arrangement of claim 14, wherein the network navigation object is adapted to display images on a computer display.*

*20. The layered arrangement of claim 14, wherein the network navigation object includes software commands for creating a datastream for transferring information between objects in the layered component computing arrangement.*

* * * * *